
COMP1406

Introduction to Computer Science II

Course Notes



Notes maintained by Mark Lanthier (2014 version)

Table of Contents

1	Programming in JAVA	1
1.1	Object-Oriented Programming and JAVA	2
1.2	Writing Your First JAVA Program	5
1.3	Python vs. Processing vs. JAVA	7
1.4	Getting User Input	11
1.5	Formatting Text	15
2	Creation and Storage of JAVA Objects	21
2.1	Using Existing JAVA Objects	22
2.2	Creating Your Own Objects in JAVA	24
2.3	Memory Allocation and Object Storage	27
3	Defining Object Behavior	39
3.1	Object Constructors (Re-Visited)	40
3.2	Defining Methods	44
3.3	Null Pointer Exceptions	52
3.4	Overloading	54
3.5	Instance vs. Class (i.e., static) Methods	56
3.6	Encapsulation - Protecting An Object's Internals	60
3.7	Changing How Objects Look When Printed	69
3.8	A Bank Example	73
4	Class Hierarchies and Inheritance	85
4.1	Organizing Classes	86
4.2	Inheritance	91
4.3	Abstract Classes & Methods	109
4.4	JAVA Interfaces	116
4.5	Polymorphism	121
5	Graphical User Interfaces	135
5.1	User Interfaces	136
5.2	Components and Containers	139
5.3	Grouping Components Together	150
5.4	Event Handling	157
6	Proper Coding Style Using MVC	173
6.1	Separate Model, View and Controller Components	174
6.2	Preparing Your Model Classes for the GUI	175
6.3	Developing a Proper View	179
6.4	Developing a Proper Controller	184
7	User Interface Extensions	191
7.1	Automatic Resizing Using Layout Managers	192
7.2	Adding Menus	209
7.3	Standard Dialog Boxes	214
7.4	Making Your Own Dialog Boxes	221

8 Abstract Data Types	237
8.1 Common Abstract Data Types	238
8.2 The List ADT	240
8.3 The Queue ADT	261
8.4 The Deque ADT	268
8.5 The Stack ADT	270
8.6 The Set ADT	276
8.7 The Dictionary / Map ADT	284
8.8 Collections Class Tools	295
8.9 Implementing an ADT (Doubly-Linked Lists)	298
9 Recursion With Data Structures	307
9.1 Recursive Efficiency	308
9.2 Examples With Self-Referencing Data Structures	310
9.3 A Maze Searching Example	330
10 Exception Handling	339
10.1 Simple Debugging	340
10.2 Exceptions	342
10.3 Examples of Handling Exceptions	352
10.4 Creating and Throwing Your Own Exceptions	359
11 Saving and Loading Information	367
11.1 Introduction to Files and Streams	368
11.2 Reading and Writing Binary Data	370
11.3 Reading and Writing Text Data	376
11.4 Reading and Writing Whole Objects	379
11.5 Saving and Loading Example	383
11.6 The File Class	390
12 Network Programming	395
12.1 Networking Basics	396
12.2 Reading Files From the Internet (URLs)	400
12.3 Client/Server Communications	404
12.4 Datagram Sockets	411
13 Other Interesting JAVA Classes	417
13.1 The String Class	418
13.2 The StringBuilder & Character Classes	424
13.3 The Date and Calendar Classes	427
14 Graphics	435
14.1 Doing Simple Graphics	436
14.2 Repainting Components	440
14.3 Displaying Images	442
14.4 Graph Editor Example	445
14.5 Adding Features to the Graph Editor	466

This page has been intentionally left blank.

Chapter 1

Programming in Java

What is in This Chapter ?

This first chapter introduces you to programming JAVA applications. It assumes that you are already familiar with programming and that you have taken either Processing or Python in the previous course. You will learn here the basics of the JAVA language syntax. In this chapter, we discuss how to make JAVA applications and the various differences between JAVA applications and the Processing/Python applications that you may be used to writing.



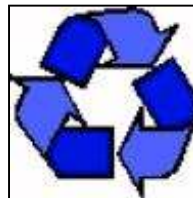
1.1 Object-Oriented Programming and JAVA

Object-oriented programming (OOP) is a way of programming in which your code is organized into objects that interact with one another to form an application. When doing OOP, the programmer (i.e., you) spends much time **defining** (i.e., writing code for) various objects by specifying the attributes (or data) that make up the object as well as small/simple functional behaviors that the object will need to respond to (e.g., deposit, withdraw, compute interest, get age, save data etc...)

There is nothing *magical* about OOP. Programmers have been coding for years in traditional top/down structured programming languages. So what is so great about OO-Programming? Well, OOP uses 3 main powerful concepts:

Inheritance

- promotes code sharing and re-usability
- intuitive hierarchical code organization



Encapsulation

- provides notion of security for objects
- reduces maintenance headaches
- more robust code



Polymorphism

- simplifies code understanding
- standardizes method naming



We will discuss these concepts later in the course once we are familiar with JAVA .

Through these powerful concepts, object-oriented code is typically:

- **easier to understand** (relates to real world objects)
- better **organized** and hence easier to work with
- **simpler** and **smaller** in size
- more **modular** (made up of plug-n'-play re-usable pieces)
- better **quality**

This leads to:

- high productivity and a **shorter delivery cycle**
- **less manpower** required
- **reduced costs** for maintenance
- more **reliable** and **robust** software
- **pluggable** systems (updated UI's, less legacy code)

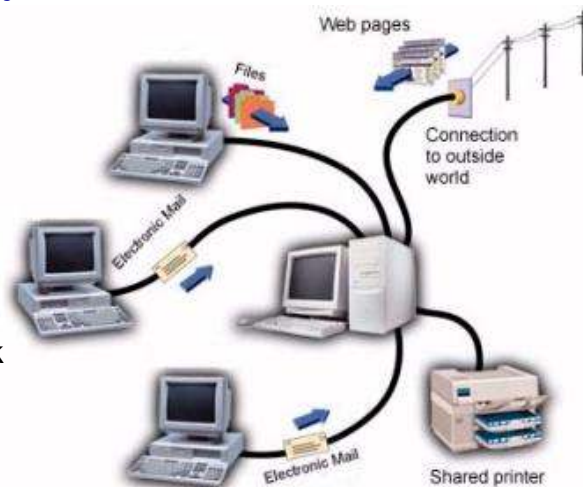
JAVA is a very popular object-oriented programming language from SUN Microsystems. In the previous course you may have used a language called **Processing** which is written "on top of" JAVA. That is, **Processing** uses the same syntax as JAVA. However **Processing** simplifies the process required to get a program "up and running" in that some of the overhead code is hidden from the programmer. In addition, some of the functionality in **Processing** has been simplified such as graphics and event handling ... which are just a little more complicated in JAVA.

JAVA has become a basis for new technologies such as: Enterprise Java Beans (EJB's), Servlets and Java Server Pages (JSPs) , etc. In addition, many packages have been added which extend the language to provide special features:

- **Java Media Framework** (for video streaming, webcams, MP3 files, etc)
- **Java 3D** (for 3D graphics)
- **Java Advanced Imaging** (for image manipulation)
- **Java Speech** (for dictation systems and speech synthesis)
- **Java FX** (for graphics, web apps, charts/forms, etc..)
- **J2ME** (for mobile devices such as cell phones)
- **Java Embedded** (for embedding java into hardware to create smart devices)

JAVA is continually changing/growing. Each new release fixes bugs and adds features. New technologies are continually being incorporated into JAVA. Many new packages are available. Just take a look at the www.oracle.com/technetwork/java/index.html website for the latest updates. There are many reasons to use JAVA:

- **architecture independence**
 - ideal for internet applications
 - code written once, runs anywhere
 - reduces cost \$\$\$
- **distributed and multi-threaded**
 - useful for internet applications
 - programs can communicate over network
- **dynamic**
 - code loaded only when needed
- **memory managed**
 - automatic memory allocation / de-allocation
 - garbage collector releases memory for unused objects
 - simpler code & less debugging
- **robust**
 - strongly typed
 - automatic bounds checking
 - no "pointers" (you will understand this in when you do **C** language programming)



The JAVA programming language itself (i.e., the SDK (Software Development Kit) that you download from SUN) actually consists of many program pieces (or object class definitions) which are organized in groups called **packages** (i.e., similar to the concept of **libraries** in other languages) which we can use in our own programs.



When programming in JAVA, you will usually use:

- classes from the JAVA class libraries (used as *tools*)
- classes that you will create yourself
- classes that other people make available to you

Using the JAVA class libraries whenever possible is a good idea since:

- the classes are carefully written and are efficient.
- it would be silly to write code that is already available to you.

We can actually create our own packages as well, but this will not be discussed in this course.

How do you get started in JAVA?

When you download and install the latest **JAVA SDK**, you will not see any particular application that you can run which will bring up a window that you can start to make programs in. That is because the SUN guys, only supply the JAVA SDK which is simply the compiler and virtual machine. JAVA programs are just text files, they can be written in any type of text editor. Using a most rudimentary approach, you can actually open up windows **NotePad** and write your program ... then compile it using the windows **Command Prompt** window. This can be tedious and annoying since JAVA programs usually require you to write and compile multiple files.

A better approach is to use an additional piece of application software called an **Integrated Development Environment (IDE)**. Such applications allow you to:

- write your code with colored/formatted text
- compile and run your code
- browse java documentation
- create user interfaces visually
- and use other java technologies (e.g. Java Beans, EJB's, Servlet programming etc...)

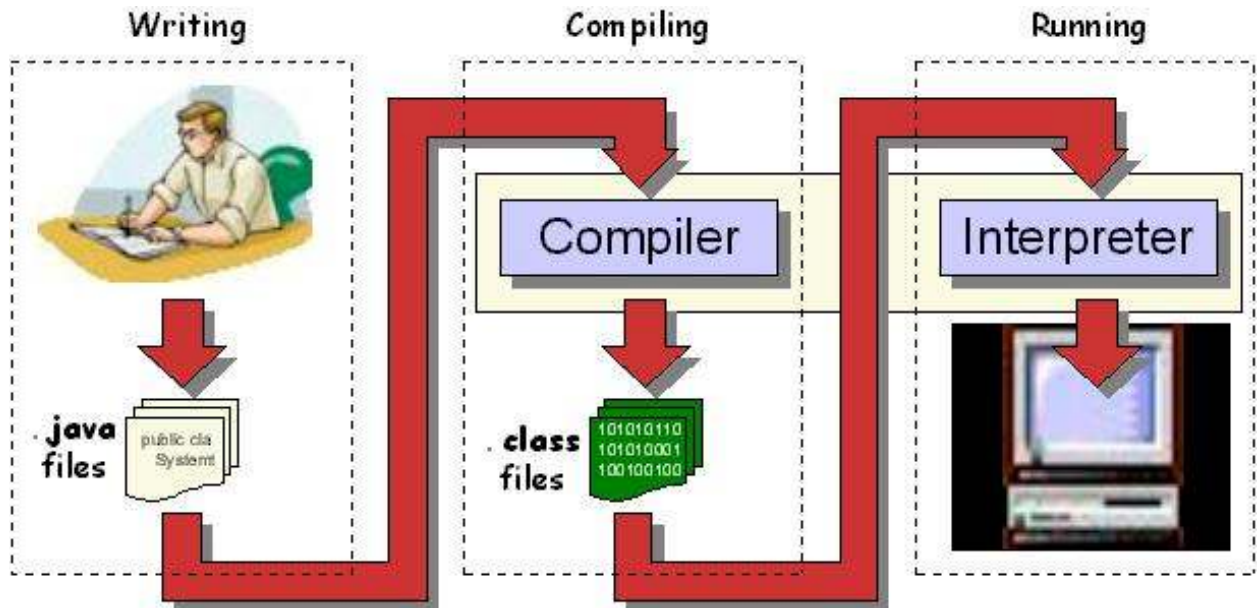
There are many IDE's that you can use. You may choose whatever you wish. Here are a few:

- **JCreator LE** (Windows) - download from www.jcreator.com
- **JGrasp** (Windows, Mac OS X, Linux) - download from www.jgrasp.com
- **Eclipse** (Windows, Mac OS X, Linux) - download from www.eclipse.org
- **Dr. Java** (Windows, Mac OS X) - download from drjava.sourceforge.net

1.2 Writing Your First JAVA Program

The process of writing and using a JAVA program is as follows:

1. **Writing:** define your classes by writing what is called **.java** files (a.k.a. **source code**).
2. **Compiling:** send these **.java** files to the JAVA compiler, which will produce **.class** files
3. **Running:** send one of these **.class** files to the JAVA interpreter to run your program.



The java **compiler**:

- prepares your program for running
- produces a **.class** file containing **byte-codes** (which is a program that is ready to run).

If there were errors during compiling (i.e., called "**compile-time**" errors), you must then fix these problems in your program and then try compiling it again.

The java **interpreter** (a.k.a. **Java Virtual Machine (JVM)**):

- is required to run any JAVA program
- reads in **.class** files (containing byte codes) and translates them into a language that the computer can understand, possibly storing data values as the program executes.

Just before running a program, JAVA uses a **class loader** to put the byte codes in the computer's memory for all the classes that will be used by the program. If the program produces errors when run (i.e., called "**run-time**" errors), then you must make changes to the program and re-compile again.

Our First Program

The first step in using any new programming language is to understand how to write a simple program. By convention, the most common program to begin with is always the "hello world" program which when run ... should output the words "Hello World" to the computer screen. We will describe how to do this now. When compared to Processing, you will notice that JAVA requires a little bit of overhead (i.e., extra code) in order to get a program to run.

All of your programs will consist of one or more files called **classes**. Last term we defined classes only to represent a data structure with some variables in it. However, in JAVA, each time you want to make any program, you need to define a class. That means, each program requires us to define a data structure (or object), although sometimes we will not even define any data (or variables) for the object.

Here is the first program that we will write:

```
public class HelloWorldProgram {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Here are a few points of interest in regards to ALL of the programs that you will write in this course:

- The program must be saved in a file with the same name as the class name (spelled the same exactly with upper/lower case letters and with a **.java** file extension). In this case, the file must be called **HelloWorldProgram.java**.
- The first line begins with words **public class** and then is followed by the name of the program (which must match the file name, except not including the .java extension). The word **public** indicates that this will be a "publically visible" class definition that we can run from anywhere. We will discuss this more later.
- The entire class is defined within the first opening brace **{** at the end of the first line and the last closing brace **}** on the last line.
- The 2nd line (i.e., **public static void main(String args[]) {**) defines the starting place for your program and will ALWAYS look exactly as shown. In Processing, the starting place for our program was simply the top line of the program and then **setup()** was called, followed by an infinite loop that called the **draw()** procedure. There are NO **setup()** or **draw()** procedures in JAVA. Instead, the program always starts running by calling this **main()** procedure which takes a String array as an incoming parameter. This String array represents what are called "command-line-arguments" which allows you to start the program with various parameters. However, we will not use these parameters in the course and so we will not discuss it further.
- The 2nd last line will be a closing brace **}**.

So ... ignoring the necessary "template" lines, the actual program consists of only one line: `System.out.println("Hello World");` which actually prints out the characters **Hello World** to the screen. You may recall that this was a little simpler in Processing since we simply did `println("Hello World");`

So ... to summarize, every java program that you will write will have the following basic format:

```
public class _____ {
    public static void main(String[] args) {
        _____
        _____
        _____
    }
}
```

Just remember that YOU get to pick the program name (e.g., **MyProgram**) which should ALWAYS start with a capital letter. Also, your code MUST be stored in a file with the same name (e.g., **MyProgram.java**). Then, you can add as many lines of code as you would like in between the inner `{ }` braces. You should ALWAYS line up ALL of your brackets using the **Tab** key on the keyboard.

In Processing, all applications ran with a graphical window that allowed us to display graphics and text as well as get user input via the keyboard and mouse. In JAVA however, there is no such window that pops up. Instead, any program output simply appears in a System console, which is usually a pane in the IDE's window.

Later in the course, we will create our own windows. For now, however, we will simply use the System console to display results. This will allow us to focus on understanding what is going on "behind the scenes" of a windowed application. It is important that we first understand the principles of Object-Oriented Programming.

1.3 Python vs. Processing vs. Java

Although **Processing** is based on **JAVA** syntax, it uses simplified names for functions and procedures. **Python** code differs even more in regards to syntax. Provided here is a brief explanation of a few of the differences (and similarities) between the three languages:

Commenting Code:

Python	Processing & JAVA (i.e., they are the same)
<code># single line comment</code>	<code>// single line comment</code>
<code>""" a multiline comment which spans more than one line. """</code>	<code>/* a multiline comment which spans more than one line. */</code>

Displaying Information To the System Console:

Python	<pre>print 'The avg is ', avg print 'All Done'</pre>
Processing	<pre>println ("The avg is " + avg); println ("All Done");</pre>
Java	<pre>System.out.println("The avg is " + avg); System.out.println("All Done");</pre>

Math Functions:

Python	Processing	Java
<code>min(a, b)</code>	<code>min(a, b)</code>	<code>Math.min(a, b)</code>
<code>max(a, b)</code>	<code>max(a, b)</code>	<code>Math.max(a, b)</code>
<code>round(a)</code>	<code>round(a)</code>	<code>Math.round(a)</code>
<code>pow(a, b)</code>	<code>pow(a, b)</code>	<code>Math.pow(a, b)</code>
<code>sqrt(a)</code>	<code>sqrt(a)</code>	<code>Math.sqrt(a)</code>
<code>abs(a)</code>	<code>abs(a)</code>	<code>Math.abs(a)</code>
<code>sin(a)</code>	<code>sin(a)</code>	<code>Math.sin(a)</code>
<code>cos(a)</code>	<code>cos(a)</code>	<code>Math.cos(a)</code>
<code>tan(a)</code>	<code>tan(a)</code>	<code>Math.tan(a)</code>
<code>degrees(r)</code>	<code>degrees(r)</code>	<code>Math.toDegrees(r)</code>
<code>radians(d)</code>	<code>radians(d)</code>	<code>Math.toRadians(d)</code>
<code>random.random()</code>	<code>random(n)</code>	<code>Math.random()</code>

Variables:

Python	Processing	Java
<code>hungry = True;</code>	<code>boolean hungry = true;</code>	<code>boolean hungry = true;</code>
<code>days = 15;</code>	<code>int days = 15;</code>	<code>int days = 15;</code>
<code>age = 19;</code>	<code>byte age = 19;</code>	<code>byte age = 19;</code>
<code>years = 3467;</code>	<code>short years = 3467;</code>	<code>short years = 3467;</code>
<code>seconds = 1710239;</code>	<code>long seconds = 1710239;</code>	<code>long seconds = 1710239;</code>
<code>gender = 'M';</code>	<code>char gender = 'M';</code>	<code>char gender = 'M';</code>
<code>amount = 21.3;</code>	<code>float amount = 21.3;</code>	<code>float amount = 21.3f;</code>
<code>weight = 165.23;</code>	<code>double weight = 165.23;</code>	<code>double weight = 165.23;</code>

Constants:

Python	Processing	Java
do not make our own by default math.pi	<code>final int DAYS = 365;</code> <code>final float RATE = 4.923;</code> PI	<code>final int DAYS = 365;</code> <code>final float RATE = 4.923f;</code> Math.PI

Type Conversion:

Python	Processing	Java
<code>d = 65.237898546;</code> <code>f = float(d);</code> <code>i = int(f);</code> <code>g = long(i);</code> <code>c = chr(i);</code>	<code>double d = 65.237898546;</code> <code>float f = (float)d;</code> <code>int i = int(f);</code> <code>float g = float(i);</code> <code>char c = char(i);</code>	<code>double d = 65.237898546;</code> <code>float f = (float)d;</code> <code>int i = (int)f;</code> <code>float g = (float)i;</code> <code>char c = (char)i;</code>

Arrays:

Python	Processing & Java (i.e., they are the same)
<code>days = zeros(30, Int)</code> <code>weights = zeros(100, Float)</code> <code>names = [];</code> <code>rentals = [];</code> <code>friends = [];</code> <code>ages = [34, 12, 45]</code> <code>weights = [4.5, 2.6, 1.5]</code> <code>names = ['Bill', 'Jen']</code>	<code>int[] days = new int[30];</code> <code>double[] weights = new double[100];</code> <code>String[] names = new String[3];</code> <code>Car[] rentals = new Car[500];</code> <code>Person[] friends = new Person[50];</code> <code>int[] ages = {34, 12, 45};</code> <code>double[] weights = {4.5, 2.6, 1.5};</code> <code>String[] names = {"Bill", "Jen"};</code>

FOR loops:

Python	Processing	Java
<code>total = 0</code> <code>for i in range (1, n):</code> <code>total += i</code> <code>print total</code>	<code>int total = 0;</code> <code>for (int i=1; i<=n; i++) {</code> <code>total += i;</code> <code>}</code> <code>println(total);</code>	<code>int total = 0;</code> <code>for (int i=1; i<=n; i++) {</code> <code>total += i;</code> <code>}</code> <code>System.out.println(total);</code>

WHILE loops:

Python	Processing	Java
<pre> speed = 0 x = 0 while x <= width: speed = speed + 2 x = x + speed </pre>	<pre> int speed = 0; int x=0; while (x <= width) { speed = speed + 2; x = x + speed; } </pre>	<pre> int speed = 0; int x=0; while (x <= width) { speed = speed + 2; x = x + speed; } </pre>

IF statements:

Python	<pre> if (grade >= 80) and (grade <=100): print 'Super!' if grade >= 50: print grade print 'Passed!' else: print 'Grade too low.' </pre>
Processing	<pre> if ((grade >= 80) && (grade <=100)) println("Super!"); if (grade >= 50) { println(grade); println("Passed!"); } else println("Grade too low."); </pre>
Java	<pre> if ((grade >= 80) && (grade <=100)) System.out.println("Super!"); if (grade >= 50) { System.out.println(grade); System.out.println("Passed!"); } else System.out.println("Grade too low."); </pre>

Procedures & Functions:

Processing	Processing and Java
<pre>def procName(x, c): // Write code here def funcName(h): result = // Write code here return result</pre>	<pre>void procName(int x, char c) { // Write code here } double funcName(float h) { result =; // Write code here return result; }</pre>

As the course continues, you will notice other differences between **Python**, **Processing** and **JAVA**. However, the underlying programming concepts remain the same. As we do coding examples throughout the course, you will get to know some of the other intricate details of basic JAVA syntax. Therefore, we will not discuss this any further at this point.

1.4 Getting User Input

In **Processing**, all applications ran with a graphical window that allowed us to display graphics and text as well as get user input via the keyboard and mouse. In **JAVA** however, there is no such window that pops up automatically.

In addition to outputting information to the console window, **JAVA** has the capability to get input from the user. Unfortunately, things are a little "messier/uglier" when getting input. The class is called **Scanner** and it is available in the `java.util` package (more on packages later).

To get input from the user, we will create a new **Scanner** *object* for input from the **System** console. Here is the line of code that gets a line of text from the user:

```
new Scanner(System.in).nextLine();
```

This line of code will wait for the user (i.e., you) to enter some text characters using the keyboard. It actually waits until you press the **Enter** key. Then, it returns to you the characters that you typed (not including the **Enter** key). You can then do something with the characters, such as print them out.

Here is a simple program that asks users for their name and then says hello to them:

```
import java.util.Scanner;    // More on this later

public class GreetingProgram {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);

        System.out.println("What is your name ?");
        System.out.println("Hello, " + keyboard.nextLine());
    }
}
```

Notice the output from this program if the letters **Mark** are entered by the user (Note that the blue text (i.e., 2nd line) was entered by the user and was not printed out by the program):

```
What is your name ?
Mark
Hello, Mark
```

As you can see, the **Scanner** portion of the code gets the input from the user and then combines the entered characters by preceding it with the "**Hello,** " string before printing to the console on the second line.

Interestingly, we can also read in integers from the keyboard as well by using the `nextInt()` function instead of `nextLine()`. For example, consider this calculator program that finds the average of three numbers entered by the user:

```
import java.util.Scanner;    // More on this later

public class CalculatorProgram {
    public static void main(String[] args) {
        int sum;

        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter three numbers:");
        sum = keyboard.nextInt() + keyboard.nextInt() + keyboard.nextInt();
        System.out.println("The average of these numbers is " + (sum/3.0));
    }
}
```


Here is the output when the **CalculatorProgram** is run with the numbers 34, 89 and 17 entered:

```
Enter three numbers:
34
89
17
The average of these numbers is 46.666666666666664
```

There is much more we can learn about the **Scanner** class. It allows for quite a bit of flexibility in reading input. In place of **nextLine()**, we could have used any one of the following to specify the kind of primitive data value that we would like to get from the user:

```
nextInt(), nextShort(), nextLong(), nextByte(), nextFloat(),
nextDouble(), nextBoolean(), next()
```

Notice that there is no **nextChar()** function available. The **next()** function actually returns a String of characters, just like **nextLine()**. If you wanted to read a single character from the keyboard (but don't forget that we still need to also press the **Enter** key), you could use the following: **next().charAt(0)**. We will look more into this later when we discuss **String** functions. It is important to use the correct function to get user input. For example, if we were to enter 10, 20 into our program above, followed by some "junk" characters ... an error will occur telling us that there was a problem with the input as follows:

```
java.util.InputMismatchException
...
at java.util.Scanner.nextInt(Unknown Source)
at BetterCalculatorProgram.main(BetterCalculatorProgram.java:11)
...
```

This is JAVA's way of telling us that something bad just happened. It is called an **Exception**. We will discuss more about this later. For now, assume that valid integers are entered.

Example:

Let us write a program that displays the following menu.

```
Luigi's Pizza
-----
                S(SML)  M(MED)  L(LRG)
1. Cheese       5.00    7.50    10.00
2. Pepperoni    5.75    8.63    11.50
3. Combination  6.50    9.75    13.00
4. Vegetarian   7.25    10.88   14.50
5. Meat Lovers  8.00    12.00   16.00
```

The program should then prompt the user for the type of pizza he/she wants to order (i.e., 1 to 5) and then the size of pizza 'S', 'M' or 'L'. Then the program should display the cost of the pizza with 13% tax added.

To begin, we need to define a class to represent the program and display the menu:

```
public class LuigisPizzaProgram {
    public static void main(String args[]) {
        System.out.println("Luigi's Pizza");
        System.out.println("-----");
        System.out.println("          S(SML)  M(MED)  L(LRG)");
        System.out.println("1. Cheese      5.00    7.50    10.00");
        System.out.println("2. Pepperoni   5.75    8.63    11.50");
        System.out.println("3. Combination 6.50    9.75    13.00");
        System.out.println("4. Vegetarian  7.25   10.88    14.50");
        System.out.println("5. Meat Lovers 8.00   12.00    16.00");
    }
}
```

We can then get the user input and store it into variables. We just need to add these lines (making sure to put `import java.util.Scanner;` at the top of the program):

```
Scanner keyboard = new Scanner(System.in);

System.out.println("What kind of pizza do you want (1-5) ?");
int kind = keyboard.nextInt();

System.out.println("What size of pizza do you want (S, M, L) ?");
char size = keyboard.next().charAt(0);
```

Now that we have the **kind** and **size**, we can compute the total cost. Notice that the cost of a small pizza increases by \$0.75 as the kind of pizza increases. Also, you may notice that the cost of a medium is 1.5 x the cost of a small and the cost of a large is 2 x a small. So we can compute the cost of any pizza based on its kind and size by using a single mathematical formula. Can you figure out the formula ?

A small pizza would cost: **smallCost = \$4.25 + (kind x \$0.75)**

A medium pizza would cost: **mediumCost =smallCost * 1.5**

A large pizza would cost: **largeCost =smallCost * 2.**

Can you write the code now ?

```
float cost = 4.25f + (kind * 0.75f);
if (size == 'M')
    cost *= 1.5f;
else if (size == 'L')
    cost *= 2;
```

And of course, we can then compute and display the cost before and after taxes. Here is the completed program:

```

import java.util.Scanner;

public class LuigisPizzaProgram {
    public static void main(String args[]) {
        System.out.println("Luigi's Pizza");
        System.out.println("-----");
        System.out.println("          S(SML)  M(MED)  L(LRG)");
        System.out.println("1. Cheese          5.00    7.50    10.00 ");
        System.out.println("2. Pepperoni       5.75    8.63    11.50 ");
        System.out.println("3. Combination     6.50    9.75    13.00 ");
        System.out.println("4. Vegetarian      7.25   10.88    14.50 ");
        System.out.println("5. Meat Lovers     8.00   12.00    16.00 ");

        Scanner keyboard = new Scanner(System.in);

        System.out.println("What kind of pizza do you want (1-5) ?");
        int kind = keyboard.nextInt();

        System.out.println("What size of pizza do you want (S, M, L) ?");
        char size = keyboard.next().charAt(0);

        float cost = 4.25f + (kind * 0.75f);
        if (size == 'M')
            cost *= 1.5f;
        else if (size == 'L')
            cost *= 2;

        System.out.println("The cost of the pizza is: $" + cost);
        System.out.println("The price with tax is: $" + cost*1.13);
    }
}

```

The above program displays the price of the pizza quite poorly. For example, here is the output of we wanted a Large Cheese pizza:

```

The cost of the pizza is: $5.0
The price with tax is: $5.6499999999999995

```

It would be nice to display money values with proper formatting (i.e., always with 2 decimal places). The next section will cover this.

1.5 Formatting Text

Consider the following similar program which asks the user for the **price** of a product, then displays the **cost** with **taxes** included, then asks for the **payment** amount and finally prints out the **change** that would be returned:

```
import java.util.Scanner;

public class ChangeCalculatorProgram {
    public static void main(String[] args) {
        // Declare the variables that we will be using
        double price, total, payment, change;

        // Get the price from the user
        System.out.println("Enter product price:");
        price = new Scanner(System.in).nextFloat();

        // Compute and display the total with 13% tax
        total = price * 1.13;
        System.out.println("Total cost:$" + total);

        // Ask for the payment amount
        System.out.println("Enter payment amount:");
        payment = new Scanner(System.in).nextFloat();

        // Compute and display the resulting change
        change = payment - total;
        System.out.println("Change:$" + change);
    }
}
```

Here is the output from running this program with a price of **\$35.99** and payment of **\$50**:

```
Enter product price:
35.99
Total cost:$40.66870172505378
Enter payment amount:
50
Change:$9.33129827494622
```

Notice all of the decimal places. This is not pretty. Even worse ...if you were to run the program and enter a price of **8.85** and payment of **10**, the output would be as follows:

```
Enter product price:
8.85
Total cost:$10.0005003888607
Enter payment amount:
10
Change:$-5.003888607006957E-4
```

The **E-4** indicates that the decimal place should be moved 4 units to the left...so the resulting change is actually **-\$0.0005003888607006957**. While the above answers are correct, it would be nice to display the numbers properly as numbers with 2 decimal places.

JAVA's **String** class has a nice function called **format()** which will allow us to format a String in almost any way that we want to. Consider (from our code above) replacing the change output line to:

```
System.out.println("Change:$" + String.format("%,1.2f", change));
```

The **String.format()** always returns a **String** object with a format that we get to specify. In our example, this **String** will represent the formatted **change** which is then printed out. Notice that the function allows us to *pass-in* two parameters (i.e., two pieces of information separated by a comma **,** character). Recall that we discussed parameters when we created constructors and methods for our own objects.

The first parameter is itself a **String** object that specifies how we want to format the resulting String. The second parameter is the value that we want to format (usually a variable name). Pay careful attention to the brackets. Clearly, **change** is the variable we want to format. Notice the format string **"%,1.2f"**. These characters have special meaning to JAVA. The **%** character indicates that there will be a parameter after the format String (i.e., the **change** variable). The **1.2f** indicates to JAVA that we want it to display the **change** as a floating point number with at least **1** digit before the decimal and exactly **2** digits after the decimal. The **,** character indicates that we would like it to automatically display commas in the money amount when necessary (e.g., \$1,500,320.28). Apply this formatting to the total amount as well:

```
import java.util.Scanner;

public class ChangeCalculatorProgram2 {
    public static void main(String[] args) {
        double price, total, payment, change;

        System.out.println("Enter product price:");
        price = new Scanner(System.in).nextFloat();

        total = price * 1.13;
        System.out.println("Total cost:$" + String.format("%,1.2f", total));

        System.out.println("Enter payment amount:");
        payment = new Scanner(System.in).nextFloat();

        change = payment - total;
        System.out.println("Change:$" + String.format("%,1.2f", change));
    }
}
```

Here is the resulting output for both test cases:

Enter product price: 35.99 Total cost:\$40.67 Enter payment amount: 50 Change:\$9.33	Enter product price: 8.85 Total cost:\$10.00 Enter payment amount: 10 Change:\$-0.00
---	---

It is a bit weird to see a value of **-0.00**, but that is a result of the calculation. Can you think of a way to adjust the **change** calculation of **payment - total** so that it eliminates the - sign ? Try it.

The **String.format()** can also be used to align text as well. For example, suppose that we wanted our program to display a receipt instead of just the change. How could we display a receipt in this format:

```

Product Price      35.99
      Tax          4.68
-----
      Subtotal     40.67
Amount Tendered   50.00
=====
      Change Due    9.33

```

If you notice, the largest line of text is the **"Amount Tendered"** line which requires 15 characters. After that, the remaining spaces and money value take up 10 characters. We can therefore see that each line of the receipt takes up 25 characters. We can then use the following format string to print out a line of text:

```
System.out.println(String.format("%15s%10.2f", aString, aFloat));
```

Here, the **%15s** indicates that we want to display a string which we want to take up exactly 15 characters. The **%10.2f** then indicates that we want to display a float value with 2 decimal places that takes up exactly 10 characters in total (including the decimal character). Notice that we then pass in two parameters: which must be a **String** and a **float** value in that order (these would likely be some variables from our program). We can then adjust our program to use this new String format as follows ...

```

import java.util.Scanner;

public class ChangeCalculatorProgram3 {
    public static void main(String[] args) {
        double price, tax, total, payment, change;

        System.out.println("Enter product price:");
        price = new Scanner(System.in).nextFloat();

        System.out.println("Enter payment amount:");
        payment = new Scanner(System.in).nextFloat();

        tax = price * 0.13;
        total = price + tax;
        change = payment - total;

        System.out.println(String.format("%15s%10.2f", "Product Price", price));
        System.out.println(String.format("%15s%10.2f", "Tax", tax));
        System.out.println("-----");
        System.out.println(String.format("%15s%10.2f", "Subtotal", total));
        System.out.println(String.format("%15s%10.2f", "Amount Tendered", payment));
        System.out.println("=====");
        System.out.println(String.format("%15s%10.2f", "Change Due", change));
    }
}

```

The result is the correct formatting that we wanted. Realize though that in the above code, we could have also left out the formatting for the 15 character strings by manually entering the necessary spaces:

```
System.out.println(String.format("  Product Price%10.2f", price));
System.out.println(String.format("                        Tax%10.2f", tax));
System.out.println("-----");
System.out.println(String.format("          Subtotal%10.2f", total));
System.out.println(String.format("Amount Tendered%10.2f", payment));
System.out.println("=====");
System.out.println(String.format("          Change Due%10.2f", change));
```

However, the **String.format** function provides much more flexibility. For example, if we used **%-15S** instead of **%15s**, we would get a left justified result (due to the **-**) and capitalized letters (due to the capital **S**) as follows:

```
PRODUCT PRICE          34.99
TAX                    4.55
-----
SUBTOTAL               39.54
AMOUNT TENDERED       50.00
=====
CHANGE DUE             10.46
```

There are many more format options that you can experiment with. Just make sure that you supply the required number of parameters. That is, you need as many parameters as you have **%** signs in your format string.

For example, the following code will produce a **MissingFormatArgumentException** since one of the arguments (i.e., values) is missing (i.e., 4 % signs in the format string, but only 3 supplied values:

```
System.out.println(String.format("$%.2f + $%.2f + $%.2f = $%.2f", x, y, z));
```



Also, you should be careful not to miss-match types, otherwise an error may occur (i.e., **IllegalFormatConversionException**).

The next page shows a table of a few other format types that you may wish to use in the future. You are not responsible for knowing or memorizing anything in that table ... it is just for your own personal use.

Hopefully, you now feel confident enough to writing simple one-file JAVA programs to interact with the user, perform some computations and solve some relatively simple problems. It would be a VERY good idea to see if you can convert some of your simpler Processing/Python programs into JAVA.

Supplemental Information (Other String.format Flags)

There are a few other format types that may be used in the format string:

Type	Description of What it Displays	Example Output
<code>%d</code>	a general integer	4096
<code>%x</code>	an integer in lowercase hexadecimal	ff
<code>%X</code>	an integer in uppercase hexadecimal	FF
<code>%o</code>	an integer in octal	377
<code>%f</code>	a floating point number with a fixed number of spaces	83.43
<code>%e</code>	an exponential floating point number	7.869877e-03
<code>%g</code>	a general floating point number with a fixed number of significant digits	0.008
<code>%s</code>	a string as given	"Hello"
<code>%S</code>	a string in uppercase	"HELLO"
<code>%n</code>	a platform-independent line end	<CR><LF>
<code>%b</code>	a boolean in lowercase	true
<code>%B</code>	a boolean in uppercase	FALSE

There are also various format flags that can be added after the `%` sign:

Format Flag	Description of What It Does	Example Output
-	numbers are to be left justified	2378.348 followed by any necessary spaces
0	leading zeros should be shown	000244.87
+	plus sign should be shown if positive number	+67.34
(enclose number in round brackets if negative	(439.67)
,	show decimal group separators	2,347,892.99

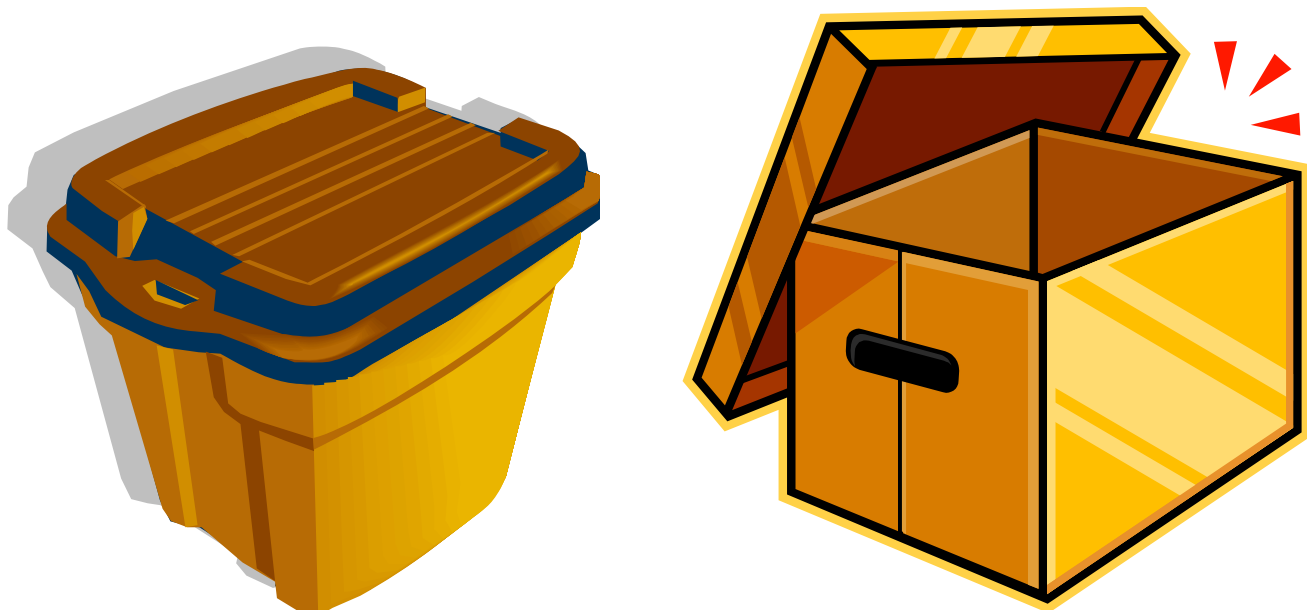
There are many options for specifying various formats including the formatting of Dates and Times, but they will not be discussed any further here. Please look at the java documentation.

Chapter 2

Creation and Storage of JAVA Objects

What is in This Chapter ?

When beginning object-oriented programming, students often have difficulty understanding how objects interact. As a result, students sometimes struggle to write code in an object-oriented manner. In this chapter we discuss how objects are created, stored and used in JAVA. In order to properly understand object-oriented programming, it is important for you to understand where data is being stored and how to access the data that is within another object. Once you understand this simple concept, your life as an object-oriented programmer will be easier. We will also discuss memory allocation so that you fully understand what an object actually is. This will help you in 2nd year when you have to allocate memory on your own.



2.1 Using Existing JAVA Objects

Until now, we have discussed creating programs by creating a class and inserting all of our code into a **main()** procedure/method. This means that our programs are considered procedural. **Object-Oriented Programming** (OOP) is *similar* to that of procedural programming in that it also involves executing a set of instructions in some specified order. However, it differs from procedural programming in the way that your code is organized.



Programming using object-oriented *style*, involves organizing your code in "chunks" that logically correspond to real-world objects. For example, you may group all of your code related to a *person* into one file (called a **class**) while code related to a *car* or a *bank account* would be grouped together in separate files (i.e., classes).

JAVA actually has a **lot** of pre-defined objects that are all organized into various **packages**. A package is essentially equivalent to a folder that contains your **.java** files. There are many standard packages in JAVA, each with many classes.

Here are just some of the standard packages that you will likely use in this course:

<code>java.lang</code>	Basic classes and interfaces required by many JAVA programs. It is automatically imported into all programs.
<code>java.util</code>	Utility classes and interfaces such as date/time manipulations, random numbers, string manipulation, collections ...
<code>java.io</code>	Classes that enable programs to input and output data.
<code>java.text</code>	Classes and interfaces for manipulating numbers, dates, characters and strings. Provides internationalization capabilities as well.

When you want to make use of some of these classes, you will use the **import** keyword to tell JAVA that you want to use a class so that it knows where to find it:

```
import <packageName>.*;
```

We did this already when we used the **Scanner** class, which is in the **java.util** package. Basically, the **import** statement is used to tell the compiler which package (i.e., directory) the class files are sitting in. You can always replace the * by a class name (where the class name is in the package) so that the readers of your code are more clear on which classes you are actually using. Keep in mind though that the **import** statement **does not load** any classes, it merely instructs the compiler where to find them when you run your code. The code is only imported/loaded by the JVM from those libraries as it is needed.

Here is a simple example that makes use of the pre-defined **Object**, **String**, **Date**, **Point** and **Rectangle** object classes in JAVA, making sure to import the correct package:

```

import java.lang.Object;
import java.lang.String;
import java.util.Date;
import java.awt.Point;
import java.awt.Rectangle;

public class ObjectTestProgram {
    public static void main(String[] args) {
        System.out.println(new Object());           // general object
        System.out.println(new String());           // blank string
        System.out.println(new Date());             // date object
        System.out.println(new Point(50, 75));      // point object
        System.out.println(new Rectangle(5,10,20,30)); // rectangle
    }
}

```

If we do not specify where to find the objects via the **import** statement, JAVA will become confused when compiling our code and will generate compile errors such as this:

```
Error: C:\...\ObjectTestProgram.java:12: cannot find symbol class Date
```

In fact, all classes in the **java.lang** package are automatically imported so we do not need the first two **import** statements. Also, when we have multiple classes being imported from the same package (e.g., **Point**, **Rectangle**), we can use a single **import** statement with the ***** wildcard character to tell JAVA to import any needed classes from that package. So here is the simplest form of the code:

```

import java.util.*;
import java.awt.*;

public class ObjectTestProgram2 {
    public static void main(String[] args) {
        System.out.println(new Object());           // general object
        System.out.println(new String());           // blank string
        System.out.println(new Date());             // date object
        System.out.println(new Point(50, 75));      // point object
        System.out.println(new Rectangle(5,10,20,30)); // rectangle object
    }
}

```

In this example, we are simply creating the objects and then displaying them. Notice how these objects are displayed in the output:

```

java.lang.Object@141ed7ac

Wed Apr 06 15:18:05 EDT 2011
java.awt.Point[x=50,y=75]
java.awt.Rectangle[x=5,y=10,width=20,height=30]

```

Each object displays itself differently. Notice that the **Date** object that was created actually corresponds to today's date and time (i.e., on April 6, 2011 when I ran the code). Also, notice that the **String** object was actually an empty string (i.e., no characters were displayed).

2.2 Creating Your Own Objects in JAVA

In the previous course, you should have already gained experience in **defining your own data structures** (a.k.a. **data types, objects**) that you used within your program in order to group various data elements together. For example, you may have created a data structure that represents someone's address as shown here.

```
class Address {
    String    name;
    int      streetNumber;
    String    streetName;
    String    city;
    String    province;
    String    postalCode;
}
```

In JAVA, we create this object by defining a **class**. Each class that we define represents a new type (or category) of object. So, the above class represents an **Address object** that we have defined. Here is a simple definition of an object as we know it so far:

*A **object** represents multiple pieces of information that are grouped together.*

A primitive data type (e.g., integer, float, character) represents a **single** simple piece of information. An object, however, is a **bundle** of data, which can be made up of multiple primitives or possibly other objects as well. You can think of an object as a bunch of small pieces of information with an elastic around it →



Once we define this class/object, then we were allowed to create **Address** objects and use them within our programs. For example, here is how we can create a new **Address** object and fill in its values:

```
Address  addr;

addr = new Address();
addr.name = "Patty O'Lantern";
addr.streetNumber = 187;
addr.streetName = "Oak St." ;
addr.city = "Ottawa";
addr.province = "ON";
addr.postalCode = "K6S8P2";

System.out.print(addr.name + " lives at ");
System.out.println(addr.streetNumber + " " + addr.streetName);
```

The code above prints out: "Patty O'Lantern lives at 187 Oak St.".

In **Processing**, each object that we made was created in the same **Processing** source code file and we were able to create variables of those types and use them within our program (as shown here on the right). However, we cannot do this in JAVA for two reasons:

1. JAVA requires ALL of our code to be defined **within a class**. So, we cannot define any variables at the top of the program like this.
2. Unlike **Processing**, there is no **setup()** or **draw()** procedure in JAVA, a **main()** procedure is used instead.

```

Car      myCar;
Car      yourCar;
House    aHouse;

class Car {
    ...
}
class House {
    ...
}

void setup() {
    ...
}
void draw() {
    ...
}

```

In JAVA, we generally define all of our own objects in separate **.java** files which will reside in the same folder as the main program class:

Even though the **Car** and **House** objects are defined in their own individual **.java** files, they cannot be **run** as programs. You can only run classes that have the **public static void main(...)** method defined. So, a JAVA program will typically consist of multiple **.java** files ... many of them being object definitions, and one of them being the actual program itself.

```

public class Car {
    ...
}
public class House {
    ...
}
public class MyProgram {
    public static void main(String[] args) {
        ...
    }
}

```

For example, we can define very simple **Car** and **Person** objects along with a test program as follows (remember that each class is defined in its own file):

```

public class Car {
    String make;
    String model;
    int year;
}

```

```

public class Person {
    String name;
    String phoneNumber;
}

```

```

public class MyObjectTestProgram {
    public static void main(String[] args) {
        System.out.println(new Car());
        System.out.println(new Person());
    }
}

```

Notice now the output from the program:

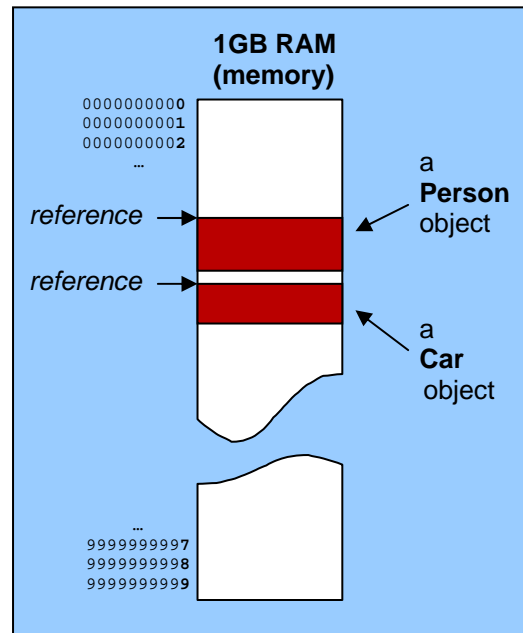
```
Car@19821f
Person@42e816
```

This is what objects look like by default. It shows the name of the class, then an @ symbol, and finally a strange combination of numbers and letters. This is what objects look like by default. They show the name of the class, then an @ symbol, and finally a strange combination of numbers and letters.

This number/letter combination represents the location (or **address**) of the object in the computer's memory. We call this the **reference**, because this memory address "refers to" the object. The actual value of the address is unimportant to us, however, it is important for you to understand that each time we make an object, it "uses up" a portion of the computer's memory.

Later we will see how to change the appearance of our objects so that they show more meaningful information when displayed.

The next section of notes will clarify in more detail exactly how these objects are stored in memory.

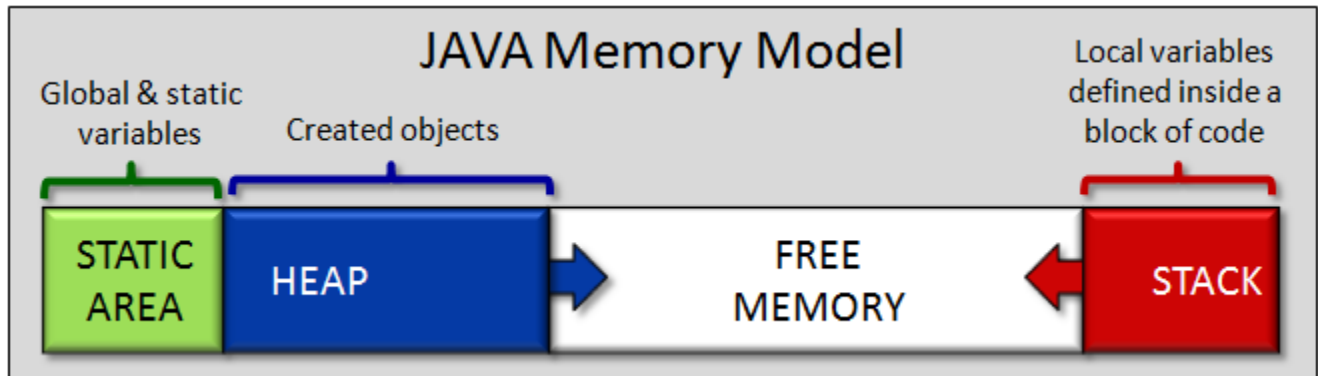


2.3 Memory Allocation and Object Storage

In order to understand how objects are stored, it is first necessary to understand how your computer's memory gets "used-up" as your program runs. The **Java Virtual Machine (JVM)** is allotted a certain amount of memory space on your computer when your program begins to run. This amount of memory allotted is adjustable via command-line arguments.

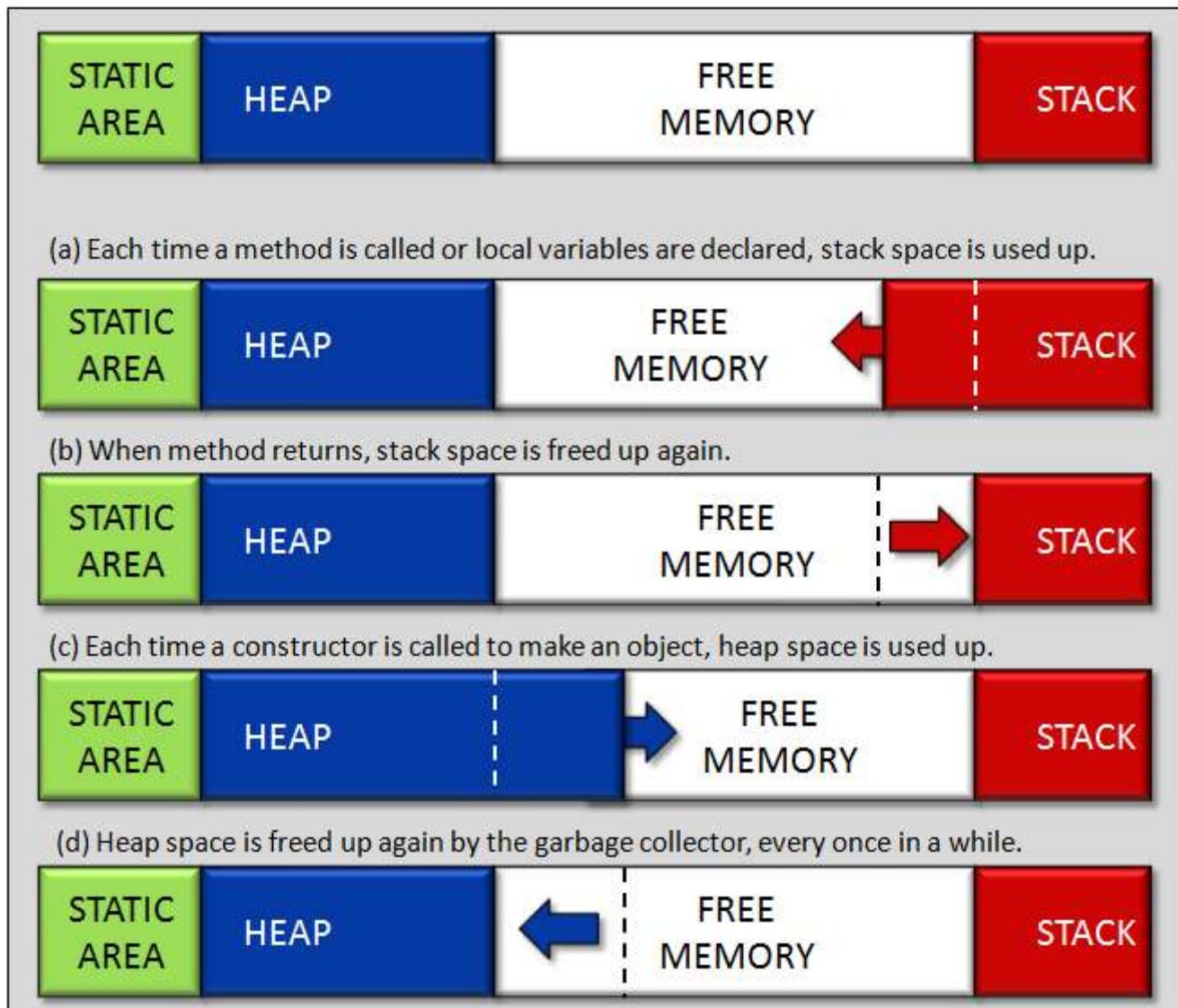
Upon start-up, some of this allotted memory is used up by the JVM. The remaining memory that is available for your program is denoted as "free memory". As your program runs, it will **allocate** (i.e., **use up**) some of this free memory at various times throughout the runtime of the program. Your program will also **return** (i.e., **free up**) this used memory at various times as it completes portions of your program. Hence the amount of available free memory will shrink and grow throughout the execution of the program.

If we consider a snapshot at any time, the memory is broken up into 4 main logical portions as shown here:



1. The **Static Area** of memory is memory that is used by the **global & static variables** that are defined by your program. This memory usage is fixed and does not change as the program runs.
2. The **Free Memory** is the memory that is not currently being used by your program. If this memory ever gets used up during your program, you will get an "Out Of Memory" error and your program will stop running.
3. The **Stack** memory is the memory that is used to store **local variables**. It also gets used up a little each time you call a method or run code within a **block of code** (i.e., a block is any code within braces). The amount of memory used during a method call depends on the number and size of the local variables defined in the method as well as its parameters.
4. The **Heap** memory is the memory that **stores all the objects** that you create. Each time that you call a constructor by using the new keyword, the **Heap** memory will increase.

The following diagram shows how the **Stack** and **Heap** memory grows and shrinks over time:



Interestingly, in JAVA, there is no way explicitly to free up heap memory from objects that you no longer want to use. The garbage collector handles this for you. You can "suggest" that the garbage collector free up memory at any time in your program by using **System.gc()**. However, this does not ensure that garbage collection will take place immediately. It is often suggested to set object-type (i.e., non primitive type) variables to **null** so that the garbage collector will realize that you are no longer holding on to an object and can free it sooner. Ultimately, the success of this strategy depends on how the garbage collector has been implemented.

For now, let us try to understand how data is stored in the stack memory.

Recall the 8 primitive data types in JAVA and the amount of memory that each requires:

Type	Bytes Used	Can Store Values Within This Range
byte	1	-128 to +127
short	2	-32,768 to +32,767
int	4	-2,147,483,648 to +2,147,483,647
long	8	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
float	4	-10 ³⁸ to +10 ³⁸
double	8	-10 ³⁰⁸ to +10 ³⁰⁸
char	2	any ASCII or UNICODE character (e.g., 'A','a','1','*','>', etc..)
boolean	1	true or false

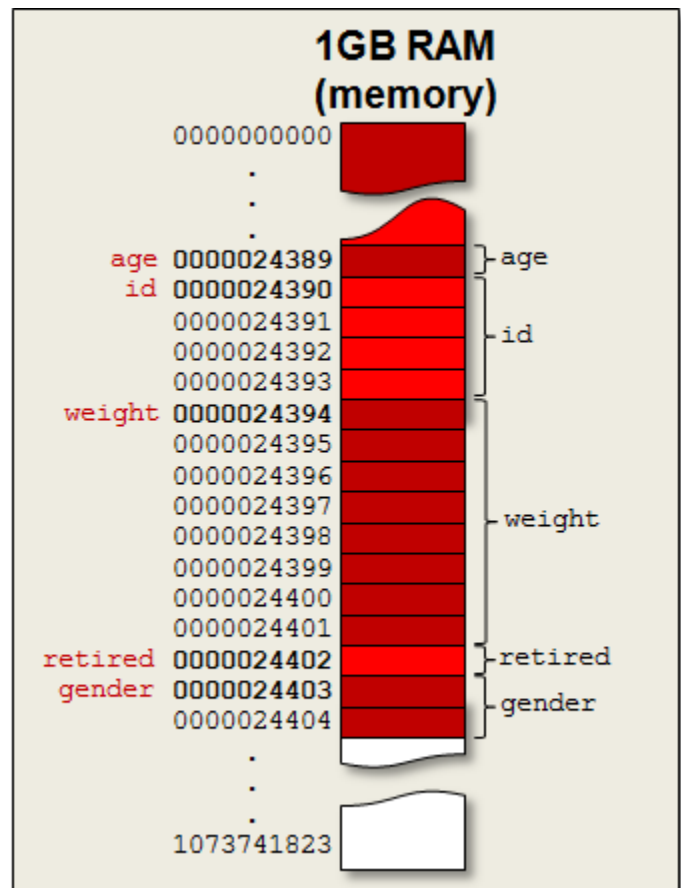
Each time we declare a variable within a method, it reserves enough space in the **STACK memory** to store the data.



For example, consider the following variables declared within a method (i.e., these are NOT object attributes) and notice the amount of memory that it consumes in the stack memory:

```
byte    age;
int     id;
double  weight;
boolean retired;
char    gender;
```

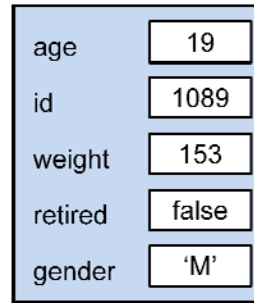
JAVA automatically reserves this space for us when we declare these variables. Each variable begins at a unique address in the computer's memory (i.e., the number shown on the left side). When using the variables, in our program, the value for the variable is obtained by simply looking at the address location to obtain the information. Similarly, when assigning values to the variables, the address is used to know where to start storing the information.



Consider now a **Person** object that stores only primitive data type attributes as follows:

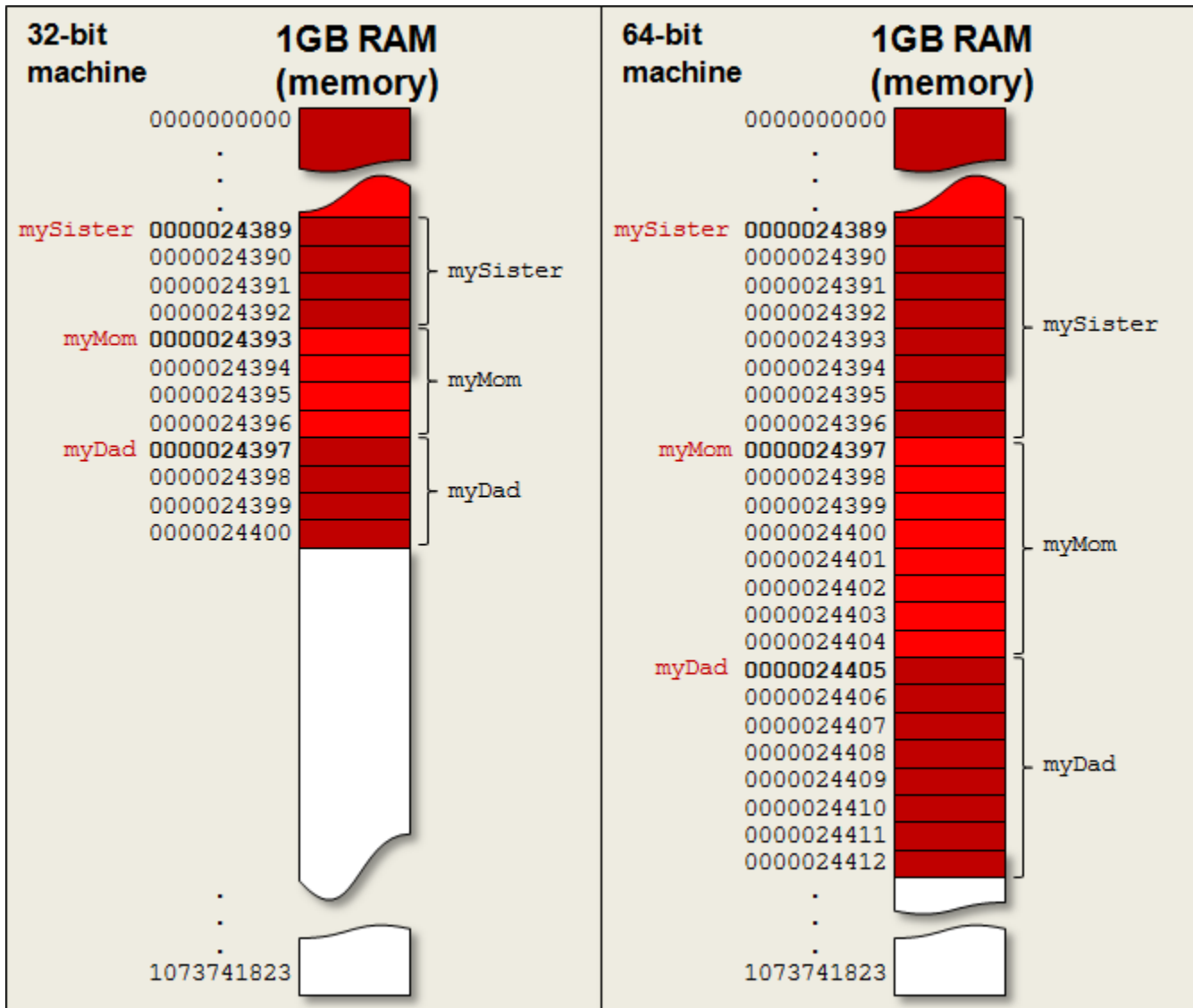
```
public class Person {
    byte    age;
    int     id;
    double  weight;
    boolean retired;
    char    gender;
}
```

a Person object



Notice how the object would be stored in memory on **32-bit** and **64-bit** machines if we were to declare a few variables of type **Person** as follows:

```
Person mySister;
Person myMom;
Person myDad;
```



Notice that on a **32-bit** machine each variable requires just **4 bytes** ... and requires **8 bytes** on a **64-bit** machine. These bytes represent *pointers* to the location in memory that the object will actually reside. A **32-bit** machine has a **32-bit** address space ... and so **4 bytes** are required to store each address reference (i.e., just the variable that holds the object ... not the object's contents). A **64-bit** machine has a **64-bit** address space ... and so each object variable requires an extra **4 byte** overhead (i.e., double the space). So **64-bit** machines, although they may be faster for CPU-related operations, may require more space allocation by default (there are ways to "compress" the pointer references...but this is not discussed here). From this point onwards in the notes, unless otherwise stated explicitly, we will assume that we are using a **32-bit** machine in order to simplify the discussion.

You will also notice that the space is not reserved for storing any of the actual data inside the object. Storing the data inside the object would require **16 bytes** of storage to store the **age** (1 byte), **id** (4 bytes), **weight** (8 bytes), **retired** (1 byte) and **gender** (2 bytes) information. However, this space is not reserved until the object is created by calling its constructor.

By declaring the variable: `Person mySister;` we get a *reference* (or pointer) to the location of the object in memory. Since we have yet to create the object ... the value of the pointer is **null**.



So, **null** actually represents an undefined memory address which requires **4 bytes** of storage at all times (**64-bit** machines require **8 bytes** to store each pointer). That means, each time that we will use objects in java, there is always a **4-byte** overhead (**8-byte** for **64-bit** machines) to store the reference to the object.

Now what about the object itself ? Consider what happens when we create the object via a constructor as follows:

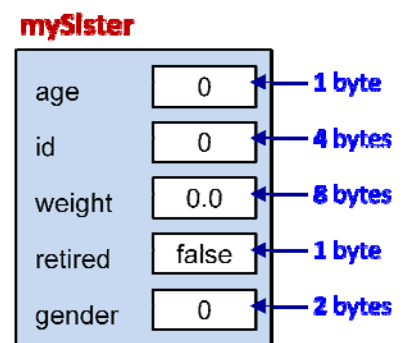
```
mySister = new Person();
```

This is now a constructor call, so the memory that will be used to store the object's data will be the **HEAP memory**:



The amount of memory used up depends on the object's data values. Looking at the class definition of the **Person** object, you will notice that it contains only primitive data types ... each of which has a fixed size.

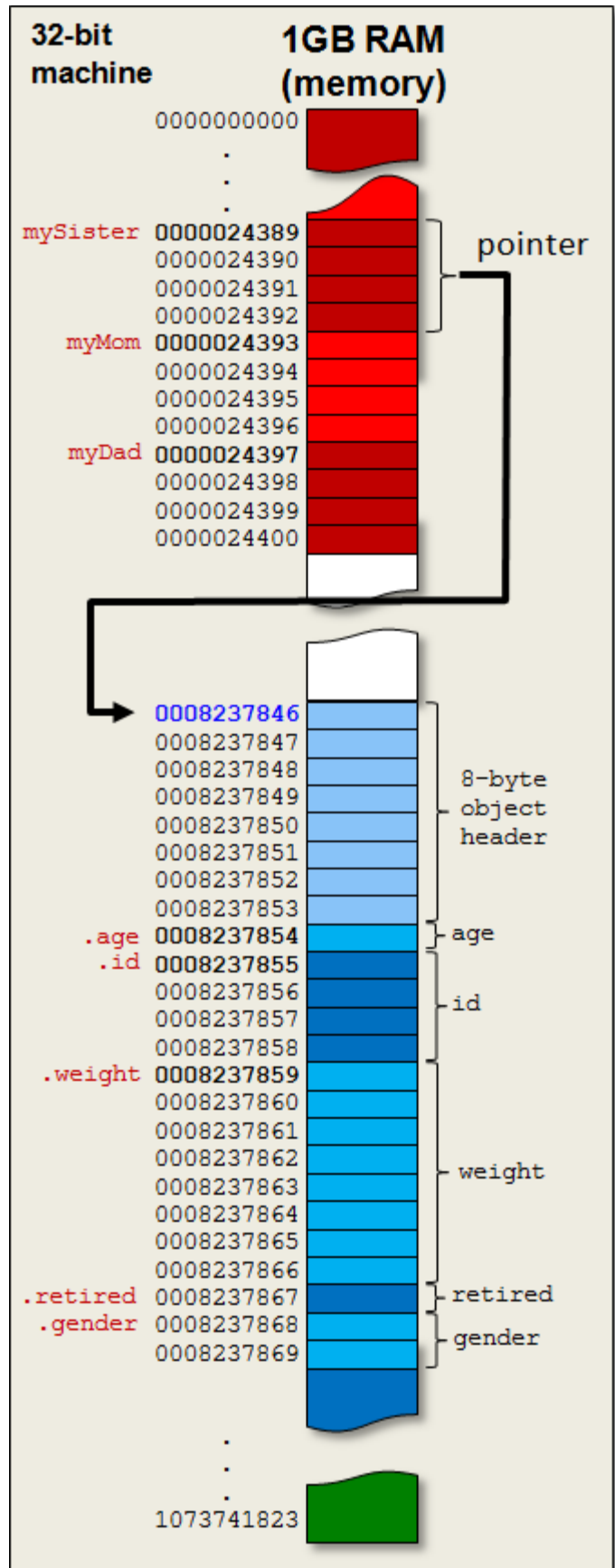
```
public class Person {
    byte    age;        // 1 byte
    int     id;         // 4 bytes
    double  weight;    // 8 bytes
    boolean retired;   // 1 byte
    char    gender;    // 2 bytes
}
```



The object requires **16 bytes** to store your data. However, each created object in JAVA requires an additional storage overhead of **8 bytes** to store an **object header**. The data contained in the header is implementation-specific ... so it depends on the particular java implementation that you are using. In fact, it is possible that other java implementations may even vary the amount of space used in the header.

Also, in some cases, additional bytes are allocated in memory to ensure that the entire object uses a multiple of **8 bytes**. That is, our current **Person** object stores **16 bytes** of data ... a nice multiple of **8**. However, if we were to add an additional **boolean** attribute to the **Person** object definition, for example, then it would take up **17 bytes**. In that case, java will probably reserve an additional **7 bytes** more to bring the total up to **24 bytes** so that the entire object again uses a multiple of **8 bytes**. These extra **7 bytes** would be unused, but nevertheless allocated.

So, each **Person** object that we create will require (16 + 8 = 24) bytes of storage as shown in the diagram here. Notice that the **mySister** variable now points to the location where the **Person** object is being stored (i.e., address 0008237846 in our example). So, the integer value of **0008237846** will be stored as the pointer at address location **0000024389**. Whenever we therefore use the **mySister** variable, JAVA just looks at its value and follows the pointer to find the object.

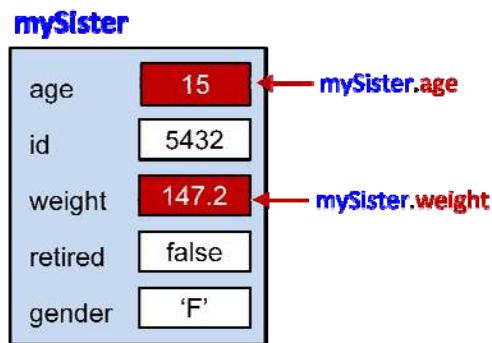


What happens when we access the internals of the **Person** object by using the dot operator ?

```

mySister.age = 15;
if (mySister.weight > 150) {
    ...
}
    
```

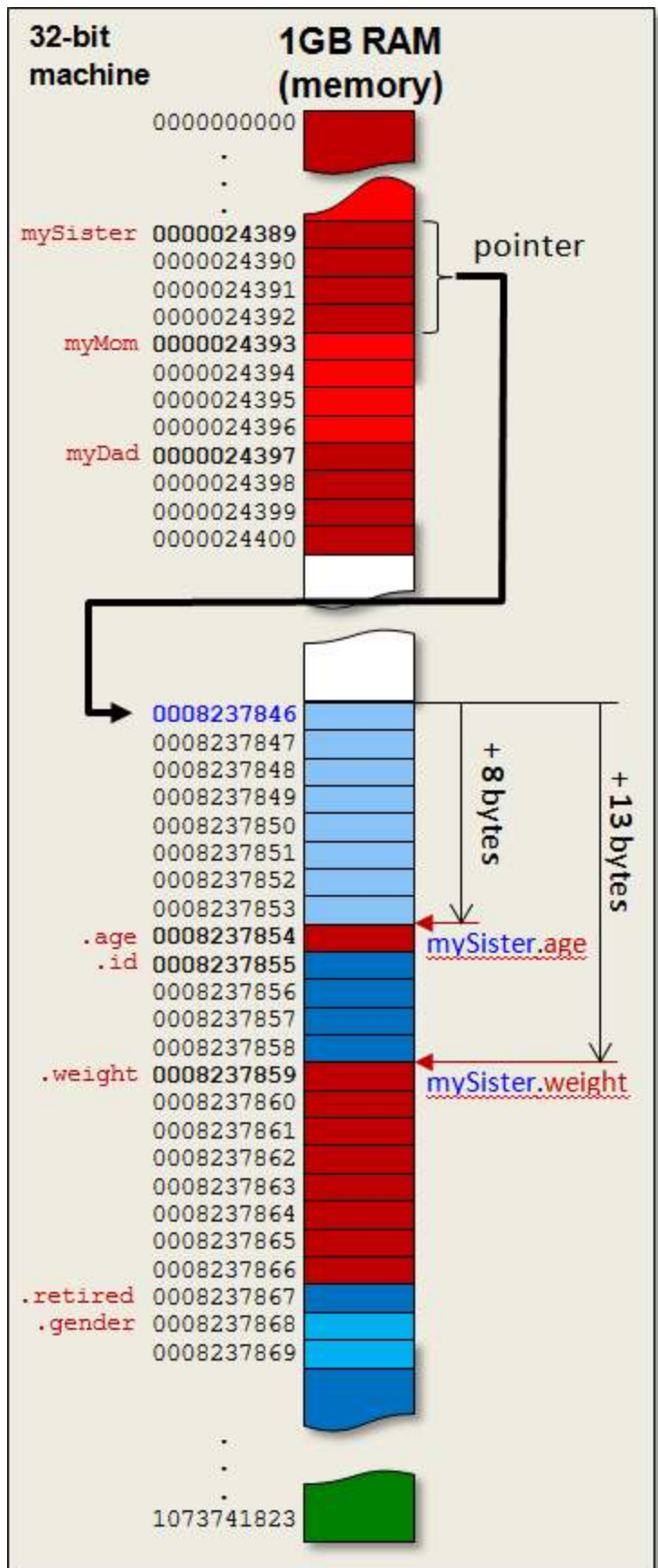
The code above requires us to go inside the object to modify its internal **age** variable/attribute and also to access the internal **weight** variable/attribute:



In order to do this, JAVA needs to determine the memory location of the **age** and **weight** attributes relative to the location of the **mySister** Person object.

JAVA begins with the address stored in the **mySister** variable (i.e., 008237846) and then adds to that value the fixed offset that the **.age** portion of the object is with respect to the start of the object (i.e., adds **8 bytes** more to bypass the header). The result is $0008237846 + 8 = 0008237854$. Once it has this location computed, it can then change the byte value there to 15 as the code instructed.

Similarly, when accessing the **.weight** portion of the object, the offset from the start of the object is $8 + 1 + 4 = 13$ bytes. Hence the weight value is found at address location $0008237846 + 13 = 0008237859$. Accessing the **double** at this address requires the 8 bytes from address 0008237859 to 0008237866 to be interpreted as a **double** value.



Now what happens when an object is contained within another object? Consider two simple objects defined as follows:

```
public class GPSLocation {
    float latitude;
    float longitude;
}

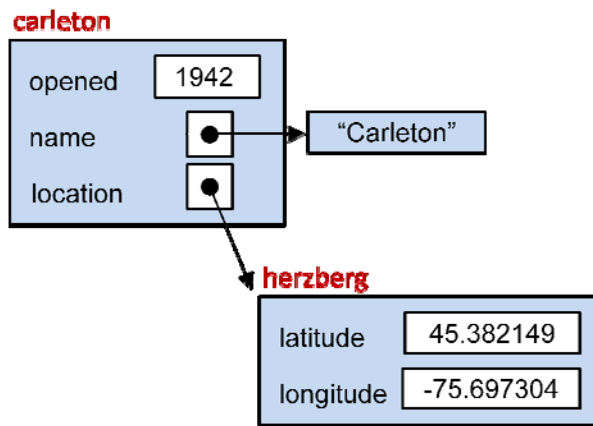
public class University {
    short opened;
    String name;
    GPSLocation location;
}
```

Now consider the following code:

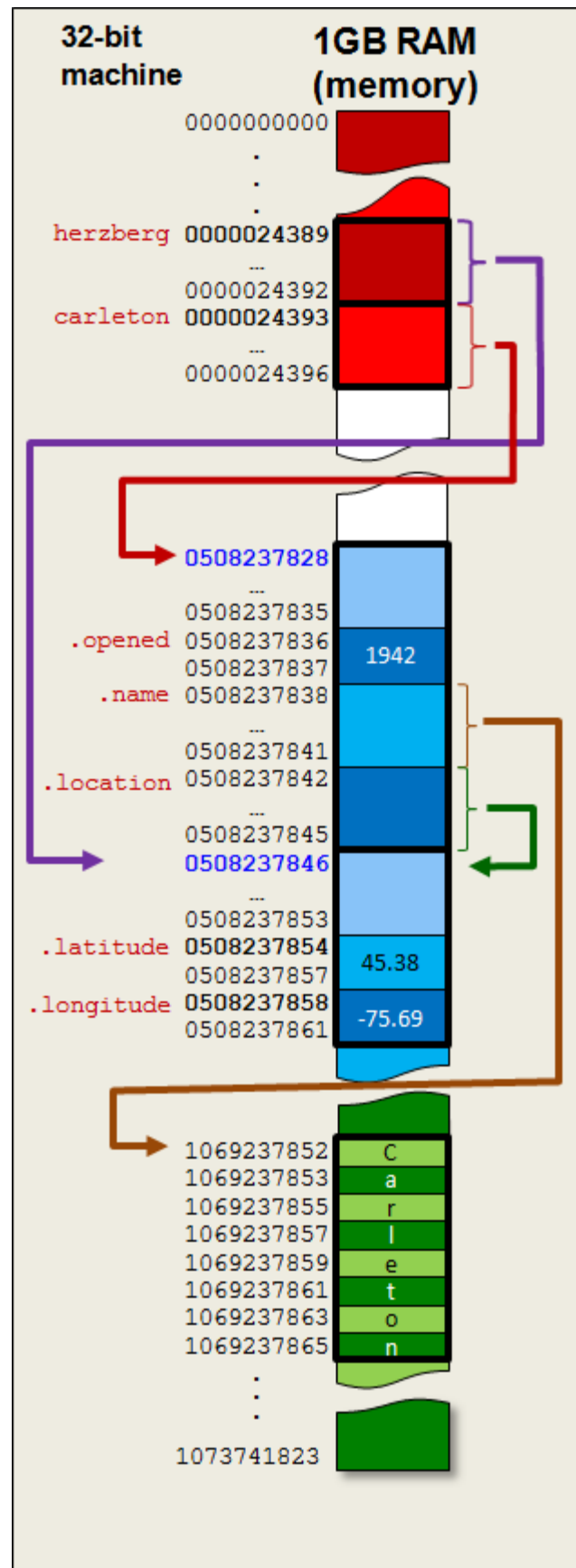
```
GPSLocation herzberg;
University carleton;

herzberg = new GPSLocation();
herzberg.latitude = 45.382149;
herzberg.longitude = -75.697304;

carleton = new University();
carleton.opened = 1942;
carleton.name = "Carleton";
carleton.location = herzberg;
```



Notice what the memory allocation will look like for this example (the picture is condensed a little vertically to fit onto the page) ---->



You can see that there are three objects involved:

- the String literal "Carleton"
- the GPSLocation
- the University

The **carleton** variable points to the **University** object ... which itself contains pointers to the **String** object ... and the **GPSLocation** object. The **herzberg** variable also points to the **GPSLocation** object and so the address value stored at locations **0000024389** and **0508237842** are the same ... which is **0508237846**.

String literals (i.e., String created using double quotes in your code) not stored in the Heap memory but are actually stored in the STATIC AREA as constants. Any other created Strings are stored in the Heap memory.

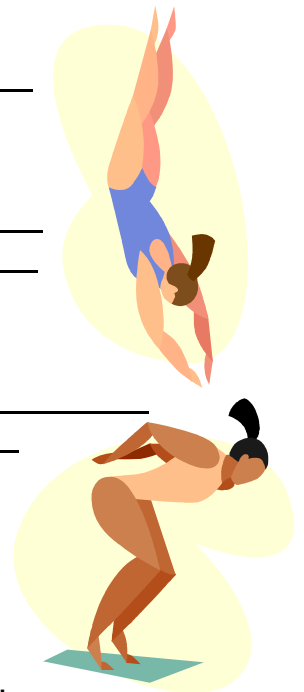
Example:

Consider writing a program that simulates a diving competition. The program will keep track of various athletes who perform dives. Assume that the following objects have been defined, each in their own files:

```
public class Dive {
    String    name;
    int      difficulty;
}
```

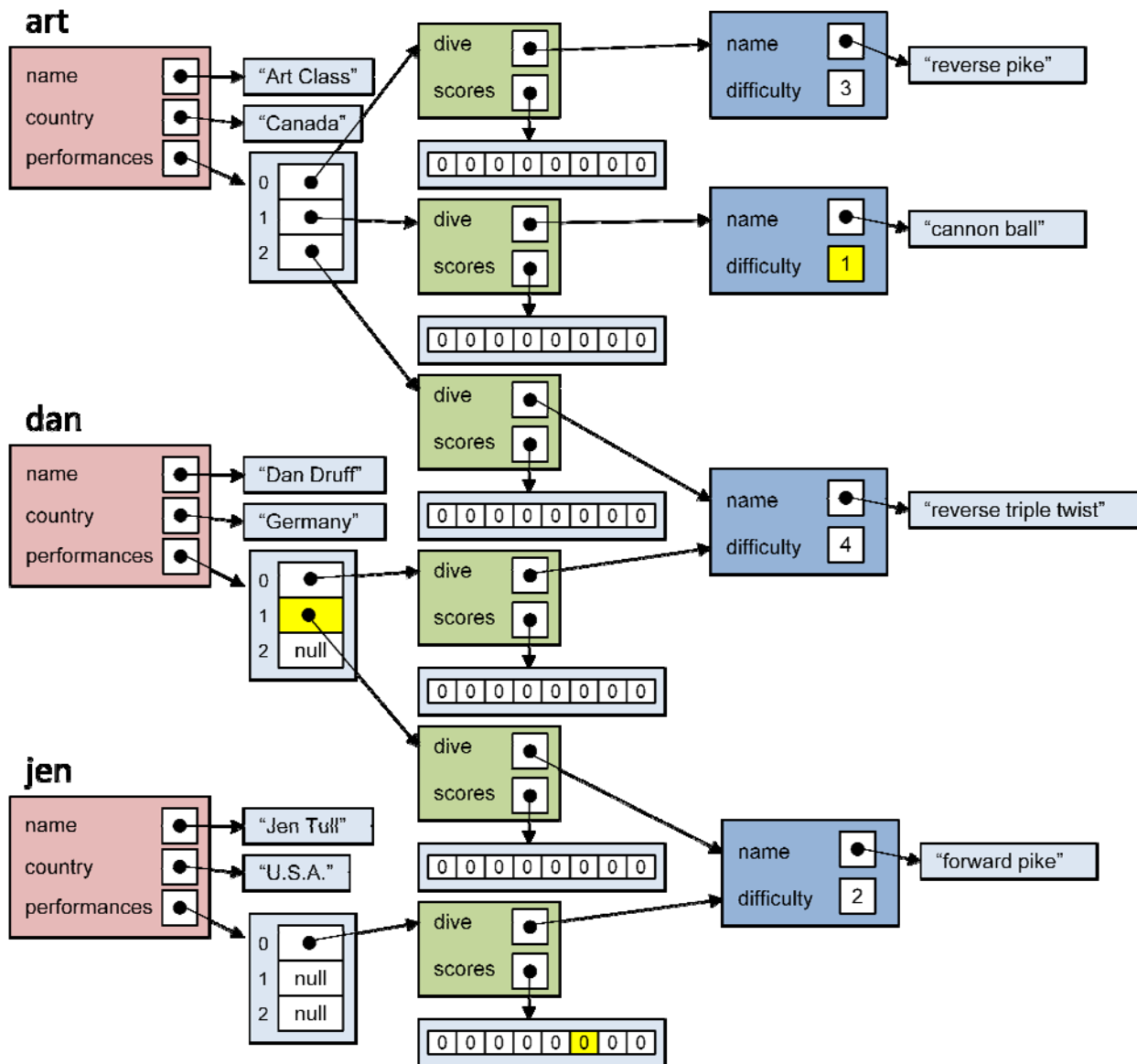
```
public class Performance {
    Dive     dive;
    float[]  scores;
}
```

```
public class Athlete {
    String    name;
    String    country;
    Performance[] performances;
}
```



Notice that each **Performance** object contains a **Dive** object. That means, each performance corresponds to a single dive (i.e., an athlete performs one dive at a time). Also, you will notice that the **Athlete** keeps an array of **Performance** objects. That is, as the athlete performs dives over time, new performances will be added to this array ... each performance representing a particular dive.

In order to make sure that we understand how objects are stored inside of one another, let us see if we can write code that constructs a particular arrangement of these objects. Here is a diagram showing the arrangement of objects that we would like to construct. Try to write the code that will produce this picture.



Looking at the picture, there are 3 **Athlete** objects, 6 **Performance** objects and 4 **Dive** objects. The remaining objects are Strings and arrays. Since the **Dive** objects don't contain the other objects that we created (i.e., neither **Athlete** nor **Performance**) we should start by making those first. We can store them into variables **d1**, **d2**, **d3** and **d4** for later use.

```
Dive d1, d2, d3, d4;
```

```
d1 = new Dive();
d1.name = "reverse pike";
d1.difficulty = 3;
```

```
d3 = new Dive();
d3.name = "reverse triple twist";
d3.difficulty = 4;
```

```
d2 = new Dive();
d2.name = "cannon ball";
d2.difficulty = 1;
```

```
d4 = new Dive();
d4.name = "forward pike";
d4.difficulty = 2;
```


Now, we should create the **Performance** objects, making sure to point them to the correct **Dive** objects.

```
Performance    p1, p2, p3, p4, p5, p6;

p1 = new Performance();           p4 = new Performance();
p1.dive = d1;                     p4.dive = d3;
p1.scores = new float[8];         p4.scores = new float[8];

p2 = new Performance();           p5 = new Performance();
p2.dive = d2;                     p5.dive = d4;
p2.scores = new float[8];         p5.scores = new float[8];

p3 = new Performance();           p6 = new Performance();
p3.dive = d3;                     p6.dive = d4;
p3.scores = new float[8];         p6.scores = new float[8];
```

Finally, we create the **Athlete** objects:

```
Athlete    art, dan, jen;

art = new Athlete();
art.name = "Art Class";
art.country = "Canada";
art.performances = new Performance[3];

dan = new Athlete();
dan.name = "Dan Druff";
dan.country = "Germany";
dan.performances = new Performance[3];

jen = new Athlete();
jen.name = "Jen Tull";
jen.country = "U.S.A.";
jen.performances = new Performance[3];
```

Of course, we need to simulate these athletes doing their performances. So we need to add the performances for each athlete:

```
art.performances[0] = p1;
art.performances[1] = p2;
art.performances[2] = p3;
dan.performances[0] = p4;
dan.performances[1] = p5;
jen.performances[0] = p6;
```

It seems like a lot of code, but we will see later how to shorten it. For now, it is just important that you understand how various objects are stored inside others.

As a further test of your understanding, see if you can write code to access and print out the following:

1. the difficulty level of Art's 2nd performance
2. the object representing Dan's 2nd performance
3. the 6th judge's score for Jen's first performance

Here are the solutions:

1. `System.out.println(art.performances[1].dive.difficulty);`
2. `System.out.println(dan.performances[1]);`
3. `System.out.println(jen.performances[0].scores[5]);`

Can you write code to determine the average judges' score for Art (consider all performances) ? Assume that there are interesting scores stored in the data, because at the moment they are all 0.

```
float    sum = 0;

for (int p=0; p<3; p++) {
    for (int s=0; s<8; s++) {
        sum = sum + art.performances[p].scores[s];
    }
}
System.out.println(sum/24);
```

Do you understand this now ? If not, you may want to come for further help during office hours. It is VERY important that you understand how to do this stuff.

Chapter 3

Defining Object Behavior

What is in This Chapter ?

This chapter discusses the basic idea behind object-oriented programming... that of **defining objects** in terms of their **state** and **behavior**. It revisits the notion of **constructors** and then explains how functions and procedures (also called **methods**) can be implemented and associated with an object's definition. The difference between **instance methods** and **static/class methods** is discussed. The notion of **encapsulation** is introduced as well as the **toString()** method that affects how an object appears when printed. The chapter concludes with a **Bank example** that makes use of 4 different objects.



3.1 Object Constructors (Re-Visited)

A constructor is a special chunk of code that we can write in our object classes that will allow us to **hide the ugliness** of setting all of the initial values for our objects each time we use them. The main advantage of making a constructor is that it will **allow us to reduce the amount of code that we need to write each time we make a new object**.

Consider, for example, a **Person** which is defined as shown below with 6 attributes:

```
public class Person {
    String    firstName;
    String    lastName;
    int       age;
    char      gender;
    boolean   retired;
    Address   address;
}
```

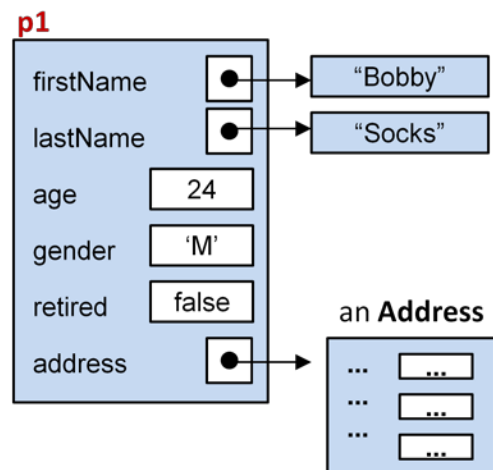
We can create a new **Person** object as follows: `new Person()`

However, to set the values for the person, we would need multiple lines of code:

```
Person    p1;

p1 = new Person();

p1.firstName = "Bobby";
p1.lastName = "Socks";
p1.age = 24;
p1.gender = 'M';
p1.retired = false;
p1.address = new Address("5 Elm St.");
```



Recall that we can write a constructor for this class that allows us to provide initial values for all of the object's attributes since a constructor is a special kind of function:

```
public Person(String f, String l, int a, char g, boolean r, Address d) {
    firstName = f;
    lastName = l;
    age = a;
    gender = g;
    retired = r;
    address = d;
}
```

Notice in JAVA, we usually indicate that a constructor is public by placing the **public** keyword in front of it. By defining the above constructor, we are thus able to specify the initial parameters for any newly-created **Person** objects as follows:

```
Person    p1, p2, p3;

p1 = new Person("Bobby", "Socks", 24, 'M', false, anAddress);
p2 = new Person("Holly", "Day", 72, 'F', true, anotherAddress);
p3 = new Person("Hank", "Urchif", 19, 'M', false, yetAnotherAddress);
```

Certainly, constructors allow us to greatly simplify our code when we need to create objects in our program.

Suppose though, that we do not know the initial parameter values to use. This would be analogous to the situation in real life where someone fills out a form but leaves some information blank. What do we do when the person leaves out information? We have two possible choices. Either **(1)** do not let them leave out any information, or **(2)** choose some kind of “default” values for the blank parts (i.e., make some assumptions by filling in something appropriate).



Up until now, we have chosen to force the user of our objects to supply parameters for ALL of the instance variables when they use (i.e., call) our constructor. So, we have taken approach number (1) above. However, in JAVA, we are allowed to create more than one constructor as long as the constructors each have a unique list of parameter types.

What if, for example, we did not know the person’s age, nor their address.

```
Person    p1, p2;

p1 = new Person("Hank", "Urchif", ?, 'M', false, ?);
p2 = new Person("Holly", "Day", ?, 'F', true, ?);
```

For this situation, we can actually define a second constructor that leaves out these two parameters:

```
public Person(String f, String l, char g, boolean r) {
    firstName = f;
    lastName = l;
    gender = g;
    retired = r;
    age = 0;
    address = null;
}
```

Notice that there are two less parameters now (i.e., no **age** and no **address**). However, you will notice that we still set the **age** and **address** to some *default* values of our choosing. What is a good default **age** and **address**? Well, we used **0** and **null**. Since we do not have an **Address** object to store in the address instance variable, we leave it undefined by setting it to **null**. Alternatively, we could have created a “dummy” **Address** object with some kind of values that would be recognizable as invalid such as:

```
address = new Address();
```

It is entirely up to you to decide what the default values should be. Make sure not to pick something that may be mistaken for valid data. For example, some bad default values for **firstName** and **lastName** would be “John” and “Doe” because there may indeed be a real person called “John Doe”.

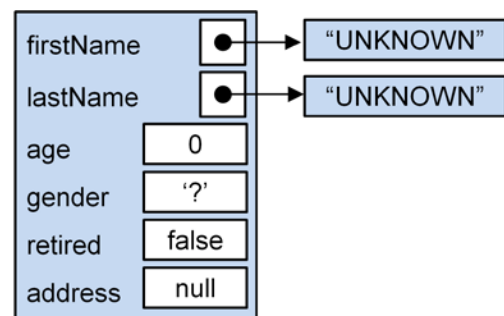
Here is one more constructor that takes no parameters. It has a special name and is known as the **zero-parameter constructor**, the **zero-argument constructor** or the **default constructor**. This time there are no parameters at all, so we need to pick default values for all the attributes:

```
public Person() {
    firstName = "UNKNOWN";
    lastName = "UNKNOWN";
    gender = '?';
    retired = false;
    age = 0;
    address = null;
}
```

You can actually create as many constructors as you want. You just need to write them all one after each other in your class definition and the user can decide which one to use at any time. Here is our resulting **Person** class definition showing the four constructors ...

```
public class Person {
    String    firstName;
    String    lastName;
    int       age;
    char      gender;
    boolean   retired;
    Address   address;

    // This is the zero-parameter constructor
    public Person() {
        firstName = "UNKNOWN";
        lastName = "UNKNOWN";
        gender = '?';
        retired = false;
        age = 0;
        address = null;
    }
}
```



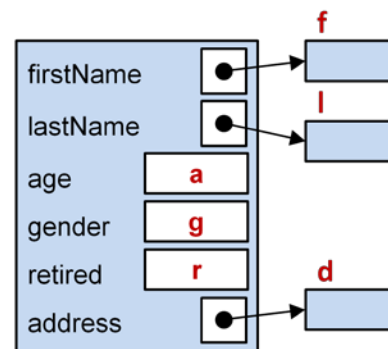
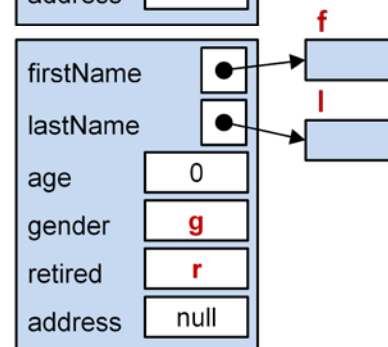
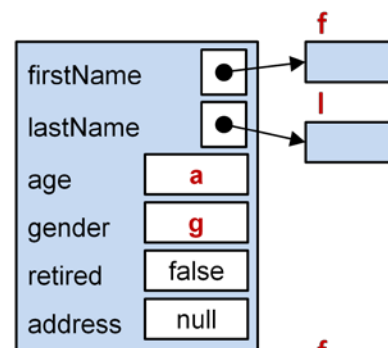
```

// This is a 4-parameter constructor
public Person(String f, String l, int a, char g) {
    firstName = f;
    lastName = l;
    age = a;
    gender = g;
    retired = false;
    address = null;
}

// This is another 4-parameter constructor
public Person(String f, String l, char g, boolean r) {
    firstName = f;
    lastName = l;
    gender = g;
    retired = r;
    age = 0;
    address = null;
}

// This is a 6-parameter constructor
public Person(String f, String l, int a, char g, boolean r, Address d) {
    firstName = f;
    lastName = l;
    age = a;
    gender = g;
    retired = r;
    address = d;
}
}

```



At any time we can use any of these constructors:

```

Person    p1, p2, p3, p4;

p1 = new Person();
p2 = new Person("Sue", "Purmann", 58, 'F');
p3 = new Person("Holly", "Day", 'F', true);
p4 = new Person("Hank", "Urchif", 19, 'M', false, new Address(...));

```

Note that it is always a good idea to ensure that you have a zero-parameter constructor. As it turns out, if you do not write any constructors, JAVA provides a zero-parameter constructor for free. That is, we can always say `new Car()`, `new Person()`, `new Address()`, `new BankAccount()` etc.. without even writing those constructors. However, once you write a constructor that has parameters, the free zero-parameter constructor is no longer available. That is, for example, if you write constructors in your `Person` class that all take one or more parameters, then you will no longer be able to use `new Person()`. JAVA will generate an error saying:

```
cannot find symbol constructor Person()
```

In general, you should always make your own zero-parameter constructor along with any others that you might like to use because others who use your class may expect there to be a zero-parameter constructor available.

3.2 Defining Methods

At this point you should understand that objects are used to group variables together in order to represent something called a data structure. Each object therefore has a set of *attributes* (also called *instance variables*) that represent the differences between members of the same class. For example, a **Vehicle** object may define a **color** attribute ... that is ... each vehicle has a color. However, that **color** value can vary from vehicle to vehicle:



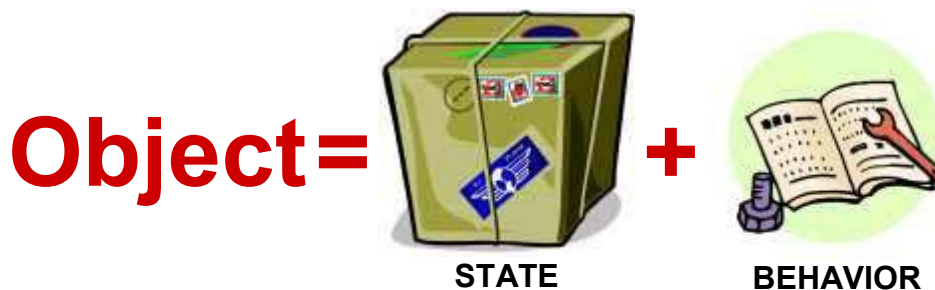
However, in real life, vehicles also vary in terms of their performance characteristics, their functionality, their abilities and their features/options:



Likewise, in object-oriented programming, in addition to defining attributes, we can also define how one particular kind of object's performance and behaviors differ from another's. Defining an object's behavior is as simple as deciding what kind of functionality that the object should have. This is nothing more than deciding which functions or procedures are required to access, modify or compute information based on the object's attributes.

Simply put ... when we define an object, we

- (1) define its attributes
- (2) define the functions and procedures that work on/with the object



What does all of this mean ? Instead of writing a single program with a list of functions/procedures, we will now be associating some of the functions/procedures with various objects. So, we will actually move some of the functions/procedures into our various class definitions.

For example, consider the **Processing** code shown here that causes two cars to accelerate across the screen. Notice how the **Car** class is used to define & maintain a car's location and speed. The **drawCar()** and **moveCar()** procedures are used to draw and move a car across the screen. The **setup()** and **draw()** procedures are essentially the code that runs the program ... showing that the two cars are continuously drawn and then moved.

Clearly, as a car moves, its location will change ... and perhaps its speed and direction ... depending on what we are really trying to simulate. Notice that the **drawCar()** and **moveCar()** procedures take (as an incoming parameter) the **Car** object that is to be drawn or moved. This is the object that gets affected by the procedure call. So, in a way, the procedure represents a *behavior* for the car, as it will affect/modify the **Car** object that is passed in.

Here is how we would define the procedures in JAVA. The **drawCar()** and **moveCar()** procedures are now written in the **Car** class ... that is ... each **Car** object now knows how to move and draw itself.

Then the main application program simply has the **setup()** and **draw()** procedures in it. Note however, that this JAVA code cannot be run, as there is no **main()** method.

Notice how the parameter is not used anymore but instead we "call" the procedure by using the car object itself, followed by the dot operator. This is similar to what we did to access an object's attributes ... because the procedures are now "inside" the **Car** object's class definition just like the attributes are defined inside there as well.

```
Car myCar, yourCar;

class Car {
  int    x, y;
  float  speed;
}

void drawCar(Car aCar) {
  ...
}

void moveCar(Car aCar) {
  ...
}

void setup() { ... }
void draw() {
  ...
  drawCar(myCar);
  drawCar(yourCar);

  moveCar(myCar);
  moveCar(yourCar);
}
```

```
public class Car {
  int    x, y;
  float  speed;

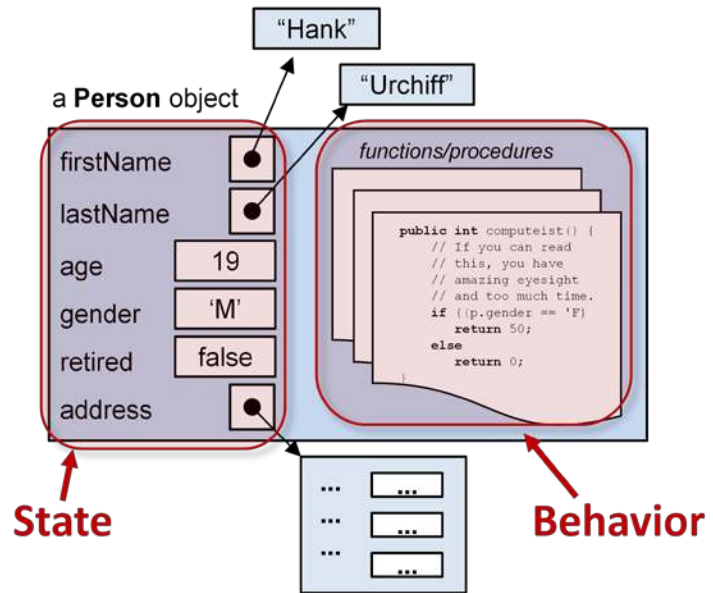
  public void drawCar() { ... }
  public void moveCar() { ... }
}
```

```
public class AccelProgram {
  Car    myCar, yourCar;

  void setup() { ... }
  void draw() {
    ...
    myCar.draw();
    yourCar.draw();

    myCar.move();
    yourCar.move();
  }
}
```

We will write most of our functions and procedures (now referred to as **methods**) within our class definitions along with the attributes. So we can think of an object as being a set of attributes (i.e., representing the object's state) as well as a set of methods (i.e., representing the object's behavior) all included "inside" the class:



Example:

Consider the **Person** class. Assume that we want to write a function that computes and returns the discount for a person who attends the theatre on "Grandma/Granddaughter Night". Assume that the discount should be 50% for women who are retired or to girls who are 12 and under. For all other people, the discount should otherwise be 0%. If we had the **Person** passed in as a parameter to the function, we could write this code:

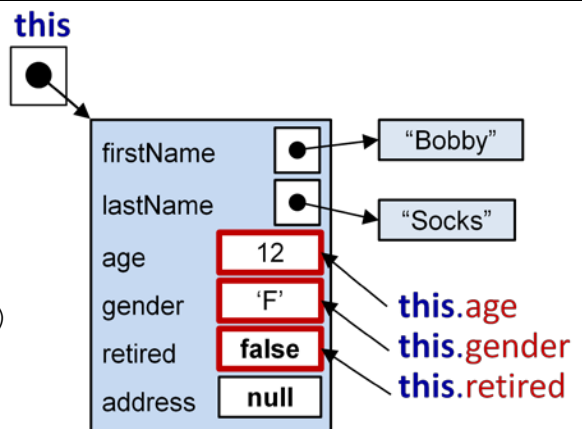
```
int computeDiscount(Person p) {
    if ((p.gender == 'F') && (p.age < 13 || p.retired))
        return 50;
    else
        return 0;
}
```



To write this as a method in JAVA, we would place this method in the **Person** class after the instance variables and constructors are defined:

```
public class Person {
    // Define attributes first
    ...
    // Now define the constructors
    ...

    // Finally, write your methods here
    public int computeDiscount() {
        if ((this.gender == 'F') &&
            (this.age < 13 || this.retired))
            return 50;
        return 0;
    }
}
```



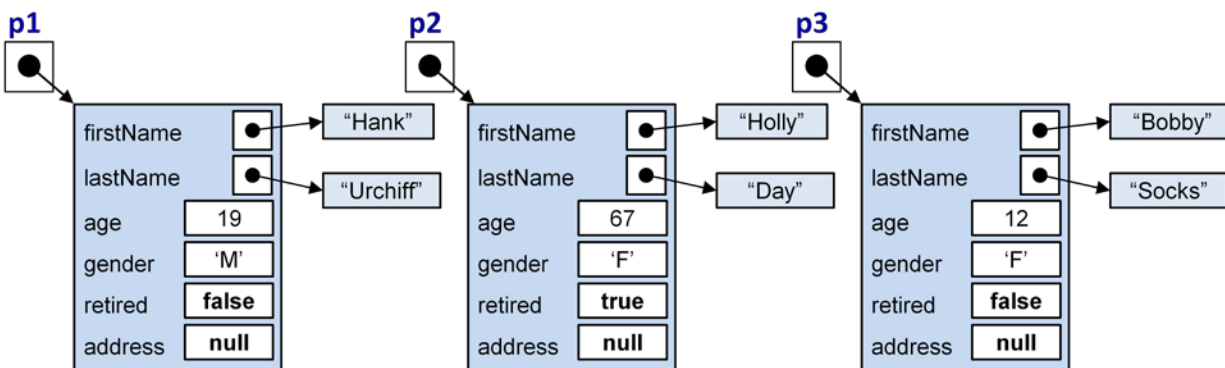
Notice that the **Person** parameter is no longer there and that the word **this** is now being used in place of that parameter. The word **this** is a special word in JAVA that can be used in methods (and constructors) to represent the object that we are applying the behavior to. That is, whatever object that we happen to call the method on, that object is represented by the word **this** within the method's body. You can think of the word **this** as being a *nickname* for the object being "worked on" within the method. Outside of the method, the word **this** is actually undefined (and therefore unusable outside of the method).

So, if we called the **computeDiscount()** method for different **Person** objects, **this** would represent the different objects **p1**, **p2** and **p3**, each time the method is called, respectively:

```
Person    p1, p2, p3;

p1 = new Person("Hank", "Urchif", 19, 'M');
p2 = new Person("Holly", "Day", 67, 'F', true, null);
p3 = new Person("Bobby", "Socks", 12, 'F');

System.out.println("p1's discount = " + p1.computeDiscount());
System.out.println("p2's discount = " + p2.computeDiscount());
System.out.println("p3's discount = " + p3.computeDiscount());
```



As it turns out, if you leave off the keyword **this**, JAVA will "assume" that you meant the object that received the method call in the first place and will act accordingly. Therefore, the following code is equivalent and often preferred since it is shorter:

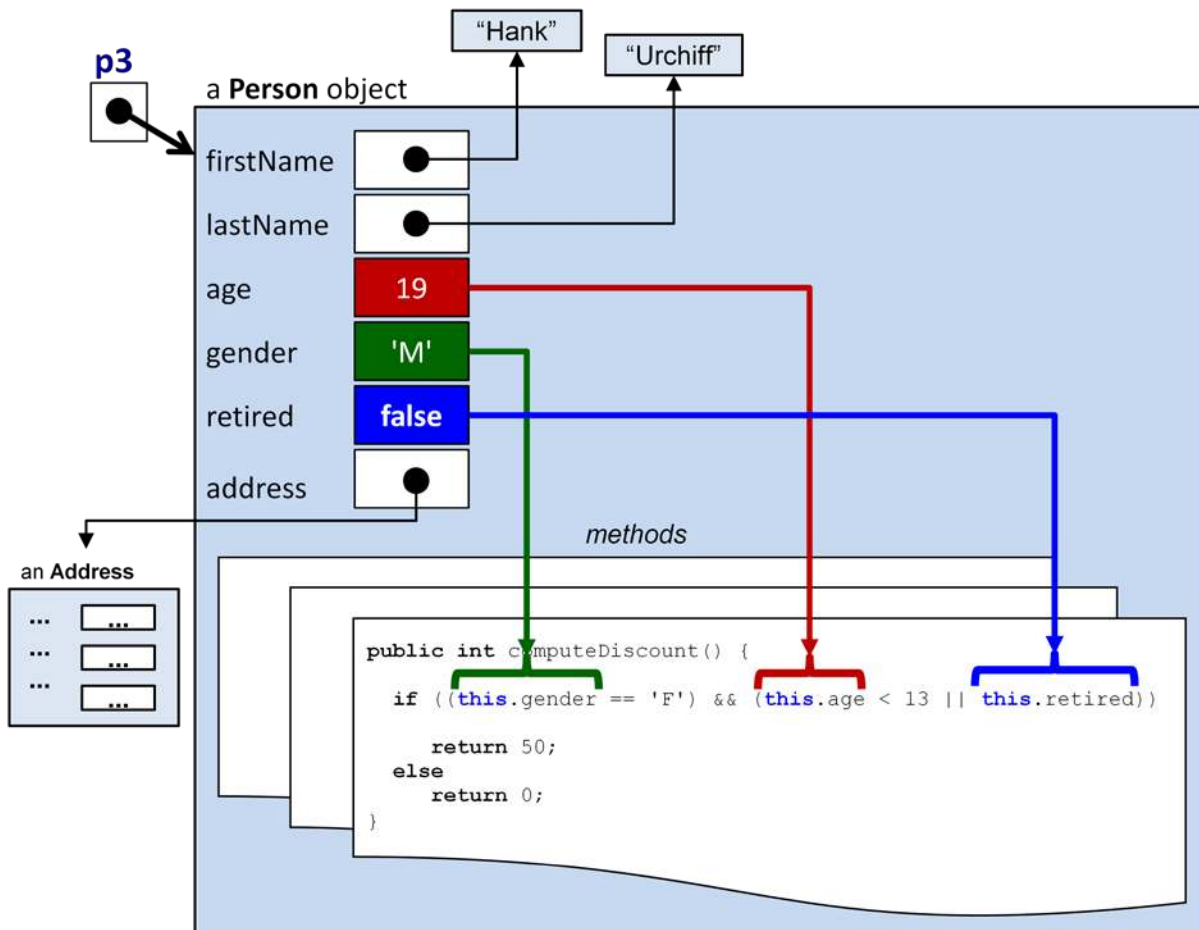
```
public class Person {
    ....

    public int computeDiscount() {
        if ((gender == 'F') && (age < 13 || retired))
            return 50;
        else
            return 0;
    }
}
```

It is important for you to understand that the **gender**, **age** and **retired** attributes are obtained from the **Person** object on which we called the **computeDiscount()** method.

You may have also noticed that the method was declared as **public**. This allows any code outside of the class to use the method.

When we test the method using `p3.computeDiscount()`... this is a picture of what is happening inside the object:



Consider writing another method that determines whether one person is older than another person. We can call the method **isOlderThan(Person x)** and have it return a **boolean** value:

```
public boolean isOlderThan(Person x) {
    if (this.age > x.age)
        return true;
    else
        return false;
}
```



... or the more efficient version:

```
public boolean isOlderThan(Person x) {
    return (this.age > x.age);
}
```

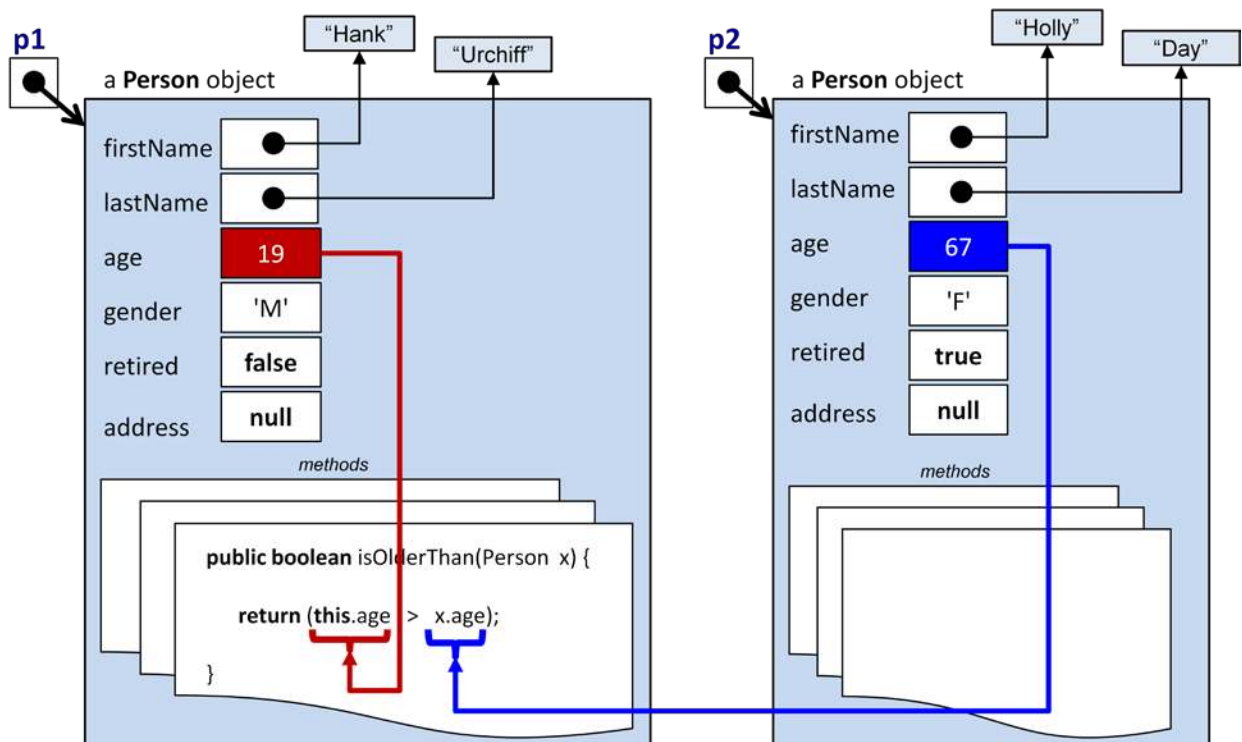
Here is a portion of a program that determines the oldest of 3 people:

```
Person    p1, p2, p3, oldest;

p1 = new Person("Hank", "Urchif", 19, 'M');
p2 = new Person("Holly", "Day", 67, 'F', true, null);
p3 = new Person("Bobby", "Socks", 12, 'F');

if (p1.isOlderThan(p2) && p1.isOlderThan(p3))
    oldest = p1;
else if (p2.isOlderThan(p1) && p2.isOlderThan(p3))
    oldest = p2;
else
    oldest = p3;
```

Consider what happens inside `p1` as we call `p1.isOlderThan(p2)`:



The method accesses the age that is stored within both **Person** objects **this** and **x**.

How could we write a similar method called **oldest()** that returns the oldest of the two **Person** objects, instead of just returning a boolean ?

```

public Person oldest(Person x) {
    if (this.age > x.age)
        return this;
    else
        return x;
}

```



Notice how the code is similar except that it now returns the **Person** object instead. Now we can simplify our program that determines the oldest of 3 people:

```

Person    p1, p2, p3, oldest;

p1 = new Person("Hank", "Urchif", 19, 'M');
p2 = new Person("Holly", "Day", 67, 'F', true, null);
p3 = new Person("Bobby", "Socks", 12, 'F');

oldest = p1.oldest(p2.oldest(p3));

```

Do you understand how this code works? Notice how the innermost **oldest()** method returns the oldest of the **p2** and **p3**. Then this oldest one is compared with **p1** in the outermost **oldest()** method call to find the oldest of the three.

In addition to writing such functions, we could write procedures that simply modify the object. For example, if we wanted to implement a **retire()** method that causes a person to retire, it would be straight forward as follows:

```

public void retire() {
    this.retired = true;
}

```

Notice that the code simply sets the retired status of the person and that the method has a void return type, indicating that there is no "answer" returned from the method's computations.

How about a method to swap the names of two people?

```

public void swapNameWith(Person x) {
    String tempName;

    // Swap the first names
    tempName = this.firstName;
    this.firstName = x.firstName;
    x.firstName = tempName;

    // Swap the last names
    tempName = this.lastName;
    this.lastName = x.lastName;
    x.lastName = tempName;
}

```



Notice how the temporary variable is required to store the **String** that is being replaced.

At this point lets step back and see what we have done. We have created 5 interesting methods (i.e., behaviors) for our **Person** object (i.e., **computeDiscount()**, **isOlderThan()**, **oldest()**, **retire()** and **swapNameWith()**). All of these methods were written one after another within the class, usually after the constructors. Here, to the right, is the structure of the class now as it contains all the methods that we wrote (the method code has been left blank to save space).

Now although these methods were **defined** in the class, they are not **used** within the class. We wrote various pieces of test code that call the methods in order to test them. Here is a more complete test program that tests all of our methods in one shot:

```
public class Person {
    // These are the instance variables
    String    firstName;
    String    lastName;
    int       age;
    char      gender;
    boolean   retired;
    Address   address;

    // These are the constructors
    public Person() { ... }
    public Person(String fn, ... ) { ... }

    // These are our methods
    int computeDiscount() { ... }
    boolean isOlderThan(Person x) { ... }
    Person oldest(Person x) { ... }
    void retire() { ... }
    void swapNameWith(Person x) { ... }
}
```

```
public class FullPersonTestProgram {
    public static void main(String args[]) {
        Person    p1, p2, p3;

        p1 = new Person("Hank", "Urchif", 19, 'M');
        p2 = new Person("Holly", "Day", 67, 'F', true, null);
        p3 = new Person("Bobby", "Socks", 12, 'F');

        System.out.println("The discount for Hank is " +
            p1.computeDiscount());
        System.out.println("The discount for Holly is " +
            p2.computeDiscount());
        System.out.println("The discount for Bobby is " +
            p3.computeDiscount());

        System.out.println("Is Hank older than Holly ? ..." +
            p1.isOlderThan(p2));
        System.out.println("The oldest person is " +
            p1.oldest(p2.oldest(p3)).firstName);

        System.out.println("Holly is retired ? ... " + p2.retired);
        p2.retire();
        System.out.println("Holly is retired ? ... " + p2.retired);
        p2.swapNameWith(p3);
        System.out.println("Holly's name is now: " +
            p2.firstName + " " + p2.lastName);
        System.out.println("Bobby's name is now: " +
            p3.firstName + " " + p3.lastName);
    }
}
```

Here is the output:

```
The discount for Hank is 0
The discount for Holly is 50
The discount for Bobby is 50
Is Hank older than Holly ? ...false
The oldest person is Holly
Holly is retired ? ... false
Holly is retired ? ... true
Holly's name is now: Bobby Socks
Bobby's name is now: Holly Day
```

3.3 Null Pointer Exceptions

In regards to calling methods, we must make sure that the object whose method we are trying to call has been through the construction process. For example, consider the following code:

```
Person    p;
System.out.println(p.computeDiscount());
```



This code will not compile. JAVA will give a compile error for the second line of code saying:

```
variable p might not have been initialized
```

JAVA is trying to tell you that you forgot to give a value to the variable **p**. In this case, we forgot to create a **Person** object.

Lets assume then that we created the **Person** as follows and then tried to get the **streetName**:

```
Person    p;
p = new Person("Hank", "Urchif", 'M', false);
System.out.println(p.address.streetName);
```

This code will now compile. Assume that the **Person** class was defined as follows:

```
public class Person {
    String    firstName;
    String    lastName;
    int       age;
    char      gender;
    boolean   retired;
    Address   address;
```



```
public Person(String fn, String ln, char g, boolean r) {
    this.firstName = fn;
    this.lastName = ln;
    this.gender = g;
    this.retired = r;
    this.age = 0;
    this.address = null;
}
...
}
```

Here the **address** attribute stores an **Address** object which is assumed to have an instance variable called **streetName**.

What will happen when we do this:

```
p.address.streetName
```

The code will generate a **java.lang.NullPointerException**. That means, JAVA is telling you that you are trying to do something with an object that was not yet defined. Whenever you get this kind of error, look at the line of code on which the error was generated. The error is always due to something in front of a dot **.** character being **null** instead of being an actual object. In our case, there are two dots in the code on that line. Therefore, either **p** is **null** or **p.address** is **null**, that is the only two possibilities. Well, we are sure that we assigned a value to **p** on the line above, so then **p.address** must be **null**. Indeed that is what has happened, as you can tell from the constructor.

To fix this, we need to do one of three things:

1. Remove the line that attempts to access the **streetName** from the address, and access it late in the program after we are sure there is an address there.
2. Check for a **null** before we try to print it and then don't print if it is **null** ... but this may not be desirable.
3. Think about why the address is **null**. Perhaps we just forgot to set it to a proper value. We can make sure that it is not **null** by giving it a proper value before we attempt to use it.

NullPointerExceptions are one of the most common errors that you will get when programming in JAVA. Most of the time, you get the error simply because you forgot to initialize a variable somewhere (i.e., you forgot to create a new object and store it in the variable).

3.4 Overloading


When we write two methods in the same class with the same name, this is called **overloading**. Overloading is only allowed if the similar-named methods have a **different** set of parameters. Normally, when we write programs we do not *think* about writing methods with the same name ... we just do it naturally. For example, imagine implementing a variety of **eat()** methods for the **Person** class as follows:

```
public void eat(Apple x) { ... }
public void eat(Orange x) { ... }
public void eat(Banana x, Banana y) { ... }
```

Notice that all the methods are called **eat()**, but that there is a variety of parameters, allowing the person to eat either an Apple, an Orange or two Banana objects. Imagine the code below somewhere in your program that calls the **eat()** method, passing in **anObject** of some type:

```
Person p;

p = new Person();
p.eat(z);
```



How does JAVA know which of the 3 **eat()** methods to call? Well, JAVA will look at what kind of object **z** actually is. If it is an **Apple** object, then it will call the 1st **eat()** method. If it is an **Orange** object, it will call the 2nd method. What if **z** is a Banana? It will NOT call the 3rd method ... because the 3rd method requires 2 Bananas and we are only passing in one. A call of **p.eat(z, z)** would call the 3rd method if **z** was a Banana. In all other cases, the JAVA compiler will give you an error stating:

```
cannot find symbol method eat(...)
```

where the **...** above is a list of the types of parameters that you are trying to use.

JAVA will NOT allow you to have two methods with the **same name AND parameter types** because it cannot distinguish between the methods when you go to use them. So, the following code will not compile:

```
public double calculatePayment(BankAccount account){...}
public double calculatePayment(BankAccount x){...}
```

You will get an error saying:

```
calculatePayment(BankAccount) is already defined in Person
```

Recall our method called **isOlderThan()** that returns a **boolean** indicating whether or not a person is older than the one passed in as a parameter:

```
public boolean isOlderThan(Person x) {
    return (this.age > x.age);
}
```

We could actually write another method in the **Person** class that took two **Person** objects as parameters:

```
public boolean isOlderThan(Person x, Person y) {
    return (this.age > x.age) && (this.age > y.age);
}
```

... and even a third method with 3 parameters:

```
public boolean isOlderThan(Person x, Person y, Person z) {
    return (this.age > x.age) && (this.age > y.age) && (this.age > z.age);
}
```

As a result, we could use any of these methods in our program:

```
Person    p1, p2, p3, p4, oldest;

p1 = new Person("Hank", "Urchif", 19, 'M');
p2 = new Person("Holly", "Day", 67, 'F');
p3 = new Person("Bobby", "Socks", 12, 'M');
p4 = new Person("Sue", "Purmann", 58, 'F');

if (p1.isOlderThan(p2,p3,p4))
    oldest = p1;
else if (p2.isOlderThan(p3,p4))
    oldest = p2;
else if (p3.isOlderThan(p4))
    oldest = p3;
else
    oldest = p4;
```

Keep in mind, however, that the parameters need not be the same type. You can have any types of parameters. Remember as well that the order makes a difference. So these would represent unique methods:

```
public int computeHealthRisk(int age, int weight, boolean smoker) { ... }
public int computeHealthRisk(boolean smoker, int age, int weight) { ... }
public int computeHealthRisk(int weight, boolean smoker, int age) { ... }
```

But these two cannot be defined together in the same class because the parameter **types** are in the same order:

```
public int computeHealthRisk(int age, int weight, boolean smoker) { ... }
public int computeHealthRisk(int weight, int age, boolean smoker) { ... }
```

3.5 Instance vs. Class (i.e., static) Methods

The methods that we have written so far have defined behaviors that worked on specific object instances. For example, when we used the `computeDiscount()` method, we did this:

```
Person    p1, p2;

p1 = new Person("Hank", "Urchif", 19, 'M');
p2 = new Person("Holly", "Day", 67, 'F');

System.out.println("p1's discount = " + p1.computeDiscount());
System.out.println("p2's discount = " + p2.computeDiscount());
```

In this example, `p1` and `p2` are variables that store instances of the `Person` class (i.e., specific individual `Person` objects). Therefore, the `computeDiscount()` method is considered to be an *instance method* of the `Person` class, since it operates on a specific *instance* of the `Person` class.

Instance methods represent behaviors (functions and procedures) that are to be performed on the particular object that we called the method for (i.e., the receiver of the method).

Instance methods typically access the inner parts of the receiver object (i.e., its attributes) and perform some calculation or change the object's attributes in some way.

A method that does not require an instance of an object to work is called a **class method**:

Class methods represent behaviors (functions and procedures) that are performed on a class ... without a particular object in mind.

Therefore, class methods do not represent a behavior to be performed on a particular receiver object. Instead, a class method represents a general function/procedure that simply happens to be located within a particular class, but does not *necessarily* have anything to do with instances of that class. Generally, class methods are not used to modify a particular instance of a class, but usually perform some computation.

For example, recall the code for the `computeDiscount()` method:

```
public int computeDiscount() {
    if ((this.gender == 'F') && (this.age < 13 || this.retired))
        return 50;
    else
        return 0;
}
```

We could have re-written the `computeDiscount()` method by supplying the appropriate `Person` object as a parameter to the method as follows:

```
public int computeDiscount(Person p) {
    if ((p.gender == 'F') && (p.age < 13 || p.retired))
        return 50;
    else
        return 0;
}
```

Notice how the method now accesses the person `p` that is passed in as the parameter, instead of the receiver `this`. If we do this, the code result is now fully-dependent on the attributes of the incoming parameter `p`, and hence independent of the receiver altogether. To call this method, we would need to pass in the person as a parameter:

```
Person    p1, p2;

p1 = new Person("Hank", "Urchif", 19, 'M');
p2 = new Person("Holly", "Day", 67, 'F', true, null);

System.out.println("p1's discount = " + p1.computeDiscount(p1));
System.out.println("p2's discount = " + p2.computeDiscount(p2));
```

We would never do this, however, since within the `computeDiscount()` method, the parameter `p` and the keyword `this` both point to the same `Person` object. So, the extra parameter is not useful since we can simply use `this` to access the person. Instead, what we probably wanted to do is to make a general function that can be written anywhere (i.e., in any class) that allows a discount to be computed for any `Person` object that was passed in as a parameter. Consider this class for example:

```
public class Toolkit {
    ...
    public int computeDiscount(Person p) {
        if ((p.gender == 'F') && (p.age < 13 || p.retired))
            return 50;
        else
            return 0;
    }
    ...
}
```

Now to call the method, we would need to make an instance of `Toolkit` as follows:

```
new Toolkit().computeDiscount(p1);
```

But this seems awkward. If we wanted to use this "tool-like" function on many people, we could do this:

```

Person    p1, p2, p3;
Toolkit   toolkit;

toolkit = new Toolkit();
p1 = ...;
p2 = ...;
p3 = ...;
System.out.println(toolkit.computeDiscount(p1));
System.out.println(toolkit.computeDiscount(p2));
System.out.println(toolkit.computeDiscount(p3));

```

Now we can see that **toolkit** is indeed a separate class from **Person** and that it acts as a container that holds on to the useful **computeDiscount()** function. However, we can simplify the code.

Anytime that we write a method that does not modify or access the attributes of an instance of the class that it is written in, the method functionality does not change. In other words, the code is not changing from instance to instance ... and is therefore considered **static**.

In our example, the code inside of the **computeDiscount()** method does not access or modify and attributes of the **Toolkit** class...it simply accesses the attributes of the **Person** passed in as a parameter as performs a computation. Therefore this method should be made **static**.

How do we do this? We simply add the **static** keyword in front of the method definition:

```

public class Toolkit {
    ...
    public static int computeDiscount(Person p) {
        if ((p.gender == 'F') && (p.age < 13 || p.retired))
            return 50;
        else
            return 0;
    }
    ...
}

```

Now we do not need to make a new **Toolkit** object in order to call the method. Instead, we simply use the **Toolkit** class name to call the method. Here is how the code changes. Notice how much simpler it is to use the method once it has been made **static**. (to save space, System.out.println has been written as **S.o.p** below):

Using it as an <i>instance</i> method	Using it as a <i>class</i> method
<pre> Person p1, p2, p3; Toolkit toolkit; toolkit = new Toolkit(); p1 = ...; p2 = ...; p3 = ...; S.o.p(toolkit.computeDiscount(p1)); S.o.p(toolkit.computeDiscount(p2)); S.o.p(toolkit.computeDiscount(p3)); </pre>	<pre> Person p1, p2, p3; p1 = ...; p2 = ...; p3 = ...; S.o.p(Toolkit.computeDiscount(p1)); S.o.p(Toolkit.computeDiscount(p2)); S.o.p(Toolkit.computeDiscount(p3)); </pre>

This is the essence of a class/static method ... the idea that the method does not necessarily need to be called by using an instance of the class.

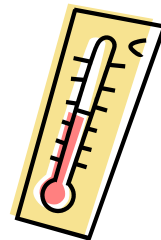
Hopefully, you will have noticed that the main difference between an instance method and a class/static method is simply in the way in which it is called. To repeat ... instance methods are called by supplying a specific instance of the object in front of the method call (i.e., a variable of the same type as the class in which the method is defined in), while class methods supply a class name in front of the method call:

```
// calling an instance method...
variableOfTypeX.instanceMethodWrittenInClassX(...);

// calling a class method...
ClassNameY.staticMethodWrittenInClassY(...);
```

Often, we use class methods to write functions that may have nothing to do with objects at all. For example, consider methods that convert a temperature value from centigrade to fahrenheit and vice-versa:

```
public static double centigradeToFahrenheit(double temp) {
    return temp * (9.0 / 5.0) + 32.0;
}
public static double fahrenheitToCentigrade(double temp) {
    return 5.0 * (temp - 32.0) / 9.0;
}
```



Where do we write such methods since they only deal with primitives, not objects? The answer is ... we can write them anywhere. We can place them at the top of the class that we would like to use them in. Or ... if these functions are to be used from multiple classes in our application, we could make another tool-like class and put them in there:

```
public class ConversionTools {
    ...
}
```

Then we could use it as follows:

```
double f = ConversionTools.centigradeToFahrenheit(18.762);
```

As you browse through the JAVA class libraries, you will notice that there are some useful static methods, however ... most methods that you will write for your own objects will be instance methods.

3.6 Encapsulation - Protecting An Object's Internals

When creating and defining an object it is a good idea to keep it simple so that anybody who uses that object in the future (including yourself) can remember how to use it. Often, there are details about an object that we don't need to know about in order to use the object. For example, when we drive a car, we need to know simple things such as:

- starting/ stopping
- steering
- changing gears
- braking, etc..



However, we do not need to worry about things such as:

- assembling the carburetor
- adjusting the spark plug timing
- installing gas lines
- changing the muffler, etc..



Cars are clearly designed to be easy to drive, requiring a simple and easy-to-understand interface. Similarly, it is important that we make our code easy to use and easy to understand. Otherwise, making changes to the code, debugging it and extending it with new features can quickly become very difficult and time consuming.

In order to keep our code simple, we need to make the interface (or "outside view") of our objects as simple as possible. That means, we need to "**hide the details**" of our object that most people would not need to worry about. That is, we need to hide some of the attributes (complicated parts) and methods (complicated procedures) for our object "under the hood", so to speak.



In addition to simplicity, there is another reason to hide some of the details of our object. We would like to prevent outsiders from "messing around with" the inner details of an object. For example we lock our car doors and trunk so that people don't get in there and take things away or damage them etc.. Similarly, for example, if we allow anyone to access the attributes of our object and perform behaviors on it in the wrong order, then this could lead to corrupt data and/or various types of errors in our code.

The idea of hiding the unnecessary details of an object and protecting inner parts of that object from general users is called *encapsulation*:

Encapsulation involves enclosing an object with a kind of "protective bubble" so that it cannot be accessed or modified without proper permission.



In JAVA, we protect and hide attributes and behaviors by using something called an **access modifier**.

*An **access modifier** is a permission setting for our attributes and methods so that they will be visible/modifiable/usable from some places in our code but not from other places.*



Access modifiers are like access levels in a high security building (e.g., no access, level 1 access, level 2 access, etc..)

By using access modifiers properly, when working with a team of software developers on a large program, some developers will have the freedom to access or modify attributes or methods from various objects, while others will not be allowed such freedom to view or change portions the objects as they would like to.

We have already been using an access modifier called **public** when we wrote our classes, constructors and various methods:

```
public class Person { ... }
public Person(String firstName, ...) { ... }
public static void main(String[] args) { ... }
public int computeDiscount() { ... }
public void deposit() { ... }
```

The keyword **public** at the front of a method declaration means that the method is publicly available to everyone, so that these methods may be called from anywhere. For most classes, constructors and methods, we do not need to write **public**. If we leave off this access modifier, then the class/constructor/method will have what is known as **default access** ... meaning that the methods may be called from any code that is in the same package or folder that this method's class is saved into. If we write all of our code in the same folder, then **default** and **public** access means the same thing.

There are two other access modifier options available called **private** and **protected**. When we declare a method as **private**, we would not be able to use this method from any class other than the class in which it is defined. **Protected** methods are methods that may be called from the method's own class or from one of its subclasses (more on this soon). So here is a summary of the access modifiers for methods:

- none - can be called from any class in the same folder
- **public** - can be called from anywhere
- **private** - can only be called from this class
- **protected** - can be called from this class or any subclasses (discussed later)

In this course, most of the methods that we write are **public** methods which allows the most freedom to access and modify our objects. Usually, **private** methods are known as **helper methods** since they are often created for the purpose of helping to reduce the size of a larger **public** method or when a piece of code is shared by several methods.

For example, consider bringing in your car for repair. The publicly available method would be called to **repair()** the car. However, many smaller sub-routines are performed as part of the repair process (e.g., **runDiagnostics()**, **disassembleEngine()** etc...). From the point of view of the user of the class, there is no need to understand the inner workings of the repair process. The user of the class may simply need to know that the car can be repaired, regardless of how it is done. Here is an example of breaking up the repair problem into *helper* methods that do the sub-routines as part of the repair ...

```
public class Car {
    public void repair() {
        this.runDiagnostics();
        this.disassembleEngine();
        this.repairBrokenParts();
        this.reassembleEngine();
        this.runDiagnostics();
    }
    private void runDiagnostics() { /*...*/ }
    private void disassembleEngine() { /*...*/ }
    private void repairBrokenParts() { /*...*/ }
    private void reassembleEngine() { /*...*/ }
}
```

Notice that the helper methods are **private** since users of this class probably do not need to call them. Here is an example showing how we might *attempt* to call these methods from some other class:

```
public class SomeCarApplicationProgram {
    public static void main(String[] args) {
        Car c = new Car();
        c.repair(); // OK to call this method
        c.disassembleEngine(); // Won't compile, since it is private
        c.repairBrokenParts(); // Won't compile, since it is private
    }
}
```

Now what about protecting an object's attributes? Well, the **public/private/protected** and *default* modifiers all work the same way as with behaviors. When used on instance variables, it allows others to be able to access/modify them according to the specified restrictions.

So far, we have never specified any modifiers for our attributes, allowing them all *default* access from classes within the same package or folder.

However, in real world situations, it is often best NOT to allow outside users to modify the internal private parts of your object. The reason is that results can often be disastrous. It is easy to relate to this because we well understand how we hide our own private parts ☺.




As an example, consider the following code, which may appear in any class. It shows that we can directly access the **balance** of a **BankAccount**.

This is clearly undesirable since there is little protection. Could you imagine if anyone could modify the balance of your bank account directly ?

```
BankAccount myAccount = new BankAccount("Mine");

myAccount.balance = 1000000.00f; // YAY
myAccount.balance = -1000000.00f; // WHY ...
```



In order to prevent direct access to important information we would need to prevent the code above from compiling/running. If we were to declare the **balance** instance variable as **private** within the **BankAccount** class, then the above code would not compile, thus solving the issue.

In general, while freedom to access/modify anything from anywhere seems like a friendly thing to do, it is certainly dangerous. Anyone could "stomp" all over our instance variables changing them at will. A general "rule-of-thumb" that should be followed is to declare ALL of your instance variables as **private** as follows:

```
public class Patient {
    private String name;
    private int age;
    private float height;
    private char gender;
    private boolean retired;

    ...
}
```

Once we do this, then the following code will not work (when written in a class other than the **Patient** class):

```
public class SomeApplicationProgram {
    public static void main(String[] args) {
        Patient p = new Patient();
        p.name = "Sandy Beach"; // will NOT compile
        p.age = 15; // will NOT compile
        p.height = 5.85f; // will NOT compile
        p.gender = 'M'; // will NOT compile
        p.retired = false; // will NOT compile
        System.out.println(p.name); // will NOT compile
        System.out.println(p.age); // will NOT compile
        System.out.println(p.height); // will NOT compile
        System.out.println(p.gender); // will NOT compile
        System.out.println(p.retired); // will NOT compile
    }
}
```

What we have essentially done is to erect a wall around the object ... like the wall around a city. We have encapsulated it with a protective bubble. Although we are still able to create the object, we are prevented from accessing or modifying its internals now from outside the class. By doing this, we have protected the object so much that we cannot get information neither into it nor out from it. We have kind of secluded the object from the rest of the world by doing this. However, just as a walled city has gates or doors to allow access, we too have a form of gated access by means of any publicly available methods.



We will grant access to "some" of our object's attributes (i.e., instance variables) by creating methods known as **get** and **set** methods (also called **getters** and **setters**). The idea of creating these gateways to our object's data is common practice and is considered to be a robust strategy when creating classes to be used in a large software application.

In this course, since we are only creating a few classes and since we are the only code writers, we may not immediately see the benefits of declaring **private** attributes and then creating these methods. However, in a larger/complicated system with hundreds of classes, the benefits become quite clear:

- object attributes are easier to understand and use
- attributes are protected from external/unknown changes
- we are following proper and robust coding style

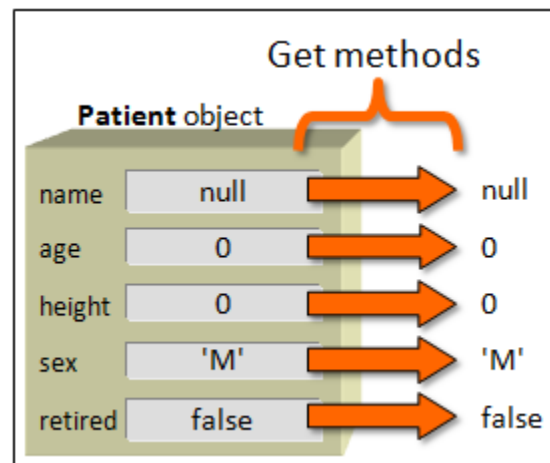
Let us first consider **get** methods. They let us look at information that is within the object by getting the object's attribute values (i.e., get the values of the instance variables). **Get** methods have:

- **public** access
- return type matching attribute's type
- name matching attribute's name
- code returning attribute's value

Here is how we would write the standard **get** methods for a **Patient** class:

```
public class Patient {
    private String name;
    private int age;
    private float height;
    private char gender;
    private boolean retired;

    // Get methods for name, age, height, gender and retired attributes
    public String getName() { return this.name; }
    public int getAge() { return this.age; }
    public float getHeight() { return this.height; }
    public char getGender() { return this.gender; }
    public boolean isRetired() { return this.retired; }
}
```



Notice that all the methods look the same in structure. They are all **public**, all have return types and names that match the attribute type, all have no parameters and all are one line long.

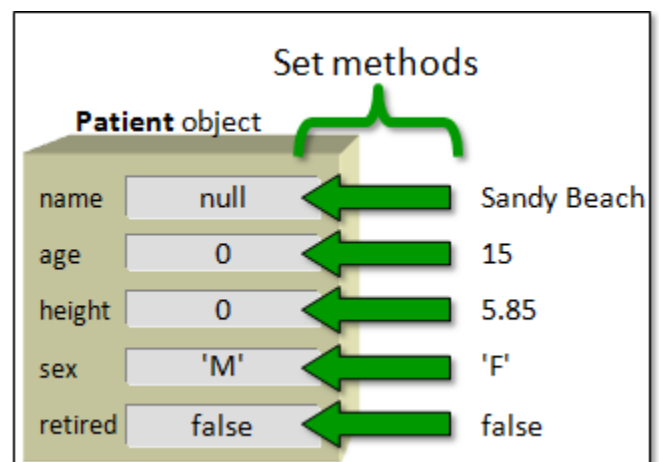
When we call the method to get the attribute value, the method simply returns the attribute value to us. It's quite simple. By convention, all get methods start with "**get**" followed by the attribute name, with the exception of attributes that are of type **boolean**. In that case, we usually use "**is**" followed by the attribute name, as it makes the method call more natural.

Now let us examine the **set** methods. **Set** methods allow us to put values into the instance variables (i.e., to set the object's attributes). **Set** methods have:

- **public** access
- **void** return type
- name matching attribute's name
- a parameter matching attribute's type
- code giving the attribute a value

Here is how we would write the standard **set** methods for the **Patient** class:

```
// Set method for name attribute
public void setName(String n) {
    this.name = n;
}
// Set method for age attribute
public void setAge(int a) {
    this.age = a;
}
// Set method for height attribute
public void setHeight(float h) {
    this.height = h;
}
// Set method for gender attribute
public void setGender(char g) {
    this.gender = g;
}
// Set method for retired attribute
public void setRetired(boolean r) {
    this.retired = r;
}
```



The single line of code in a **set** method is quite simple also.

When we call the method to give the attribute a new value (i.e., we supply the new value as a parameter to the method), the method simply takes that new attribute value and sets the attribute to it by using the = operator.

Normally, we write all the **get** and **set** methods together, and sometimes shorten them onto one line. Also, they are often listed in the code right after the **public** constructors as follows:

```

public class Patient {
    private String    name;
    private int      age;
    private float    height;
    private char     gender;
    private boolean  retired;

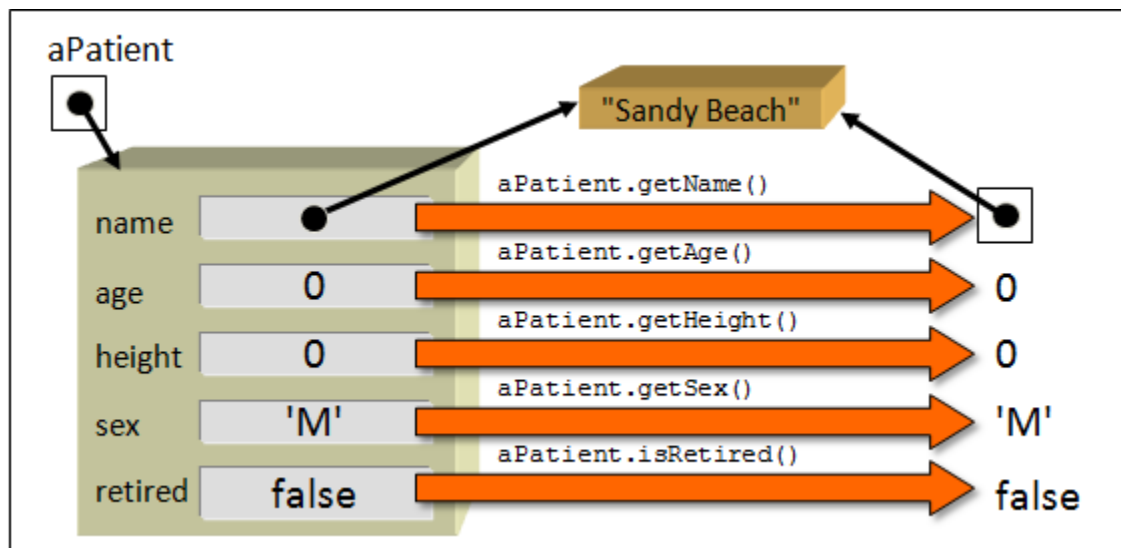
    // Constructor
    public Patient() {
        name = "Unknown";
        age = 0;
        height = 0;
        gender = '?';
        retired = false;
    }

    // Get methods
    public String getName() { return this.name; }
    public int   getAge()   { return this.age; }
    public float getHeight() { return this.height; }
    public char  getGender() { return this.gender; }
    public boolean isRetired() { return this.retired; }

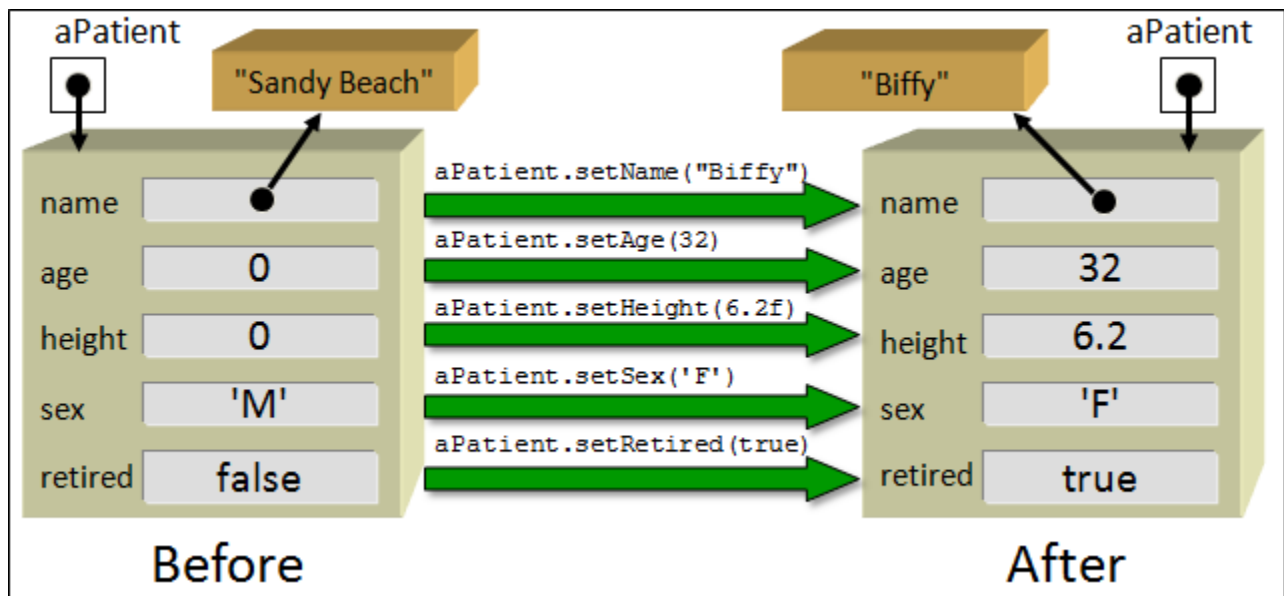
    // Set methods
    public void setName(String n) { this.name = n; }
    public void setAge(int a) { this.age = a; }
    public void setHeight(float h) { this.height = h; }
    public void setGender(char g) { this.gender = g; }
    public void setRetired(boolean r) { this.retired = r; }
}

```

Here is how the **get** method works:



Here is how the **set** method works:



Notice that primitive attribute values are simply replaced with the new value. For object attribute values, after the **set** call, the attribute will point to the new object. The previous object that the attribute used to point to is discarded (i.e., garbage collected) if no other objects are holding on to it. Once we create these **get/set** methods, we can then access and modify the object from anywhere in our program as before:

```
public class TestPatientProgram {
    public static void main(String[] args) {
        Patient p = new Patient();

        System.out.println("Before Setting ...");
        System.out.println(p.getName()); // was println(p.name);
        System.out.println(p.getAge()); // was println(p.age);
        System.out.println(p.getHeight()); // was println(p.height);
        System.out.println(p.getGender()); // was println(p.gender);
        System.out.println(p.isRetired()); // was println(p.retired);

        p.setName("Sandy Beach"); // was p.name = "Sandy Beach";
        p.setAge(15); // was p.age = 15;
        p.setHeight(5.85f); // was p.height = 5.85f;
        p.setGender('F'); // was p.gender = 'F';
        p.setRetired(true); // was p.retired = true;

        System.out.println("\nAfter Setting ...");
        System.out.println(p.getName()); // was println(p.name);
        System.out.println(p.getAge()); // was println(p.age);
        System.out.println(p.getHeight()); // was println(p.height);
        System.out.println(p.getGender()); // was println(p.gender);
        System.out.println(p.isRetired()); // was println(p.retired);
    }
}
```

Here is what the output would be (however, initial values depend on the **Patient** constructor):

```
Before Setting ...
Unknown
0
0.0
?
false

After Setting ...
Sandy Beach
15
5.85
F
true
```

Now if we think for a moment ... what did we really do by making all the **get** and **set** methods? Really, we wrote a lot of code (e.g., 5 **get** methods and 5 **set** methods for the **Patient** class) but did not gain anything new. The code does the same thing as before. In fact, the test code seems longer and perhaps slower (since we are calling a method to get/set the instance variables for us instead of accessing them directly). So why did we do this? Let us review the advantages again:

1. First, **get/set** methods actually make life simpler for users of your class because the user does not have to understand the “guts” of the object being used. It allows them to treat the object as a “black box”. The user does not need to know about all the instance variables. Some are used to hold data that is temporary or private. You should only create public **get** methods for the instance variables that the user of the class would need to know about.
2. Second, it prevents the users of a class from directly modifying the object's internals. Recall, for example, that we should never be able to directly change the balance of our bank account without going through the proper transaction procedures such as depositing and withdrawal. Of course, if we always create **public get/set** methods for all our attributes, then we still would have no such protection. So, it is important to create **set** methods **only** for the attributes that you want the user of the class to be able to change directly. Therefore, you do not always need to make **set** methods.



3.7 Changing How Objects Look When Printed

As just described, properly-designed model classes will use *encapsulation* to hide any unnecessary information from those who will make use of those classes and to keep things simple. Sometimes however, it is desirable to be able to visually distinguish one object from another. For example, consider the following code:

```
public class MyObjectTestProgram {
    public static void main(String[] args) {
        System.out.println(new Patient()); // a patient object
        System.out.println(new Patient()); // another patient object
    }
}
```

The result on the screen is as follows:

```
Patient@7d8a992f
Patient@164f1d0d
```

By default, JAVA displays all of the objects that you make in this manner, showing you the type of object (i.e., the class name) followed by something that represents the object's location in memory. This format for displaying objects is not very useful for debugging. If we had a dozen or so **Patient** objects displayed in this manner, we would not be able to "pick out" one that we may be looking for. It would be more advantageous if we had something a little more descriptive ... perhaps showing the patient's name.

What JAVA happens to be doing here is converting the **Patient** object to a **String** object first and then displaying the resulting characters to the screen. In fact, every object in JAVA has, by default, a method called **toString()** which will convert the object to a **String**.

The **Strings** returned from the call to **toString()** have the exact same characters that are displayed when we just display the objects directly using **System.out.println()**. That is because whenever JAVA attempts to display anything to the console, it automatically calls the **toString()** method for the object to convert it to characters before displaying. So, the two lines shown below do exactly the same thing:

```
Patient p = new Patient();

System.out.println(p); // displays Patient@7d8a992f
System.out.println(p.toString()); // displays Patient@7d8a992f
```



Why do we care? Well, we can actually replace the default **toString()** behavior by writing our own **toString()** method for all of our own objects that defines exactly how to convert our object to a **String**. That is, we can control the way our object “looks” when we print it on the screen or when we display it in our User Interface.

Suppose that we want our **Patient** object to display something like this when printed:

```
Patient named Hank
```

You should notice that the first two words of this output are fixed and it is only the last part (i.e., the first name of the **Patient**) that varies from patient to patient. We can make this to be the standard output format for all **Patient** objects simply by writing the following method in the **Patient** class:

```
public String toString() {  
    return ("Patient named " + this.name);  
}
```

This method overrides the default **toString()** method, essentially replacing it. Notice that the method is called **toString()** with no parameters and that it has a return type of **String**. This is important in order for the method to properly override the one inherited from the **Object** class.

Consider the output of the following code:

```
Patient          p1, p2, p3;  
  
p1 = new Patient(); // assume first name is set to "" within constructor  
p2 = new Patient();  
p2.setName("Holly");  
p3 = new Patient();  
p3.setName("Hank");  
  
System.out.println(p1);  
System.out.println(p2);  
System.out.println(p3);
```

Here is the output ...

```
Patient named  
Patient named Holly  
Patient named Hank
```

Now what if we wanted the output to be in this format instead:

```
19 year old Patient named Hank
```

To write an appropriate **toString()** method, we need to understand what is fixed in this output and what will vary. The number **19** should vary for each patient as well as the **first** and **last** names. Here is how we could write the code (replacing our previous **toString()** method):

```
public String toString() {
    return (this.age + " year old Patient named " + this.name);
}
```

Notice that the basic idea behind creating a **toString()** method is to simply keep joining together **String** pieces to form the resulting **String**. Now here is a harder one. Let us see if we can make it into this format:

19 year old non-retired patient named Hank

Here we have the **age** and **names** being variable again but now we also have the added variance of their **retirement status**.

Here is one attempt:

```
public String toString() {
    return (this.age + " year old " + this.retired + " patient named "
        + this.name);
}
```

However, this is not quite correct. This would be the format we would end up with:

19 year old false patient named Hank

Notice that we cannot simply display the value of the **retired** attribute but instead need to write **“retired”** or **“non-retired”** for the **retired** status.

To do this then, we will need to use an **IF** statement. However, in JAVA, we cannot write an **IF** statement in the middle of a **return** statement. So we will need to do this using more than one line of code. We can make an **answer** variable to hold the result and then break down our method into logical pieces that append to this **answer**:

```
public String toString() {
    String answer;

    answer = this.age + " year old ";
    answer = answer + this.retired;
    answer = answer + " patient named " + this.name);

    return answer;
}
```

Now we can insert the appropriate **IF** statements as follows:

```
public String toString() {
    String answer;

    answer = this.age + " year old ";

    if (this.retired)
        answer = answer + "retired";
    else
        answer = answer + "non-retired";
    answer = answer + " patient named " + this.name;

    return answer;
}
```

The result is what we wanted. Note however, that we can simplify this code a little further:

```
public String toString() {
    String answer = this.age + " year old ";

    if (!this.retired)
        answer = answer + "non-";

    return (answer + "retired patient named " + this.name);
}
```

3.8 A Bank Example

Consider implementing some software for a bank. Likely we need a **Bank** object that will contain **BankAccount** objects where each account is owned by a bank **Customer**. So, we will need a few interacting objects. Let's begin with a **Customer** object. We can define it as a simpler version to a **Person** object with get/set methods and a **toString()** method as follows:

```
public class Customer {
    private String    firstName;
    private String    lastName;
    private Address   address;
    private String    phoneNumber;

    // This is the zero-parameter constructor
    public Customer() {
        firstName = "UNKNOWN";
        lastName = "UNKNOWN";
        address = null;
        phoneNumber = "(???)??-????";
    }

    // This is a 4-parameter constructor
    public Customer (String f, String l, Address a, String p) {
        firstName = f;
        lastName = l;
        address = a;
        phoneNumber = p;
    }

    // These are the get/set methods
    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public Address getAddress() { return address; }
    public String getPhoneNumber() { return phoneNumber; }
    public void setFirstName(String s) { firstName = s; }
    public void setLastName(String s) { lastName = s; }
    public void setAddress(Address a) { address = a; }
    public void setPhoneNumber(String p) { phoneNumber = p; }

    // This returns a String representation of the customer
    public String toString() {
        return "Customer: " + firstName + " " + lastName +
            " living at " + address;
    }
}
```

Of course, we will need to make the **Address** object too. We can make something quite simple like this (we will leave off city/province/postal code to keep things simple):

```

public class Address {
    private String    streetNumber;
    private String    streetName;

    // This is the 2-parameter constructor
    public Address(String number, String name) {
        streetNumber = number;
        streetName = name;
    }

    // These are the get/set methods
    public String getStreetNumber() { return streetNumber; }
    public String getStreetName() { return streetName; }
    public void setStreetName(String s) { streetName = s; }
    public void setStreetNumber(String s) { streetNumber = s; }

    // This returns a String representation of the address
    public String toString() {
        return streetNumber + " " + streetName;
    }
}

```

Now, let us define a **BankAccount** with the following attributes and constructors as follows:

```

public class BankAccount {
    private Customer  owner;
    private int       accountNumber;
    private float     balance;
}

```

Likely, when someone makes a new **BankAccount**, they DO NOT get to choose their own account number, as this is usually assigned by the bank itself. Let us assume that the first created account is assigned the account number 100001, the second gets 100002, the third 100003 and so on. In this scenario, we can simply keep a counter that starts at 100001 and increases each time a new account is created.

To do this, we can create a static/class variable in the **BankAccount** class to represent this counter. We can call it **LAST_ACCOUNT_NUMBER** which will store the account number that was last given out. We can give this variable an initial value of 100000 as follows ...

```

private static int LAST_ACCOUNT_NUMBER = 100000;

```

Then, when a new **BankAccount** is created, we can give it an **accountNumber** which is one more than the **LAST_ACCOUNT_NUMBER** and then increment this counter to get it ready for the next time. This counter of ours will work exactly like one of those ticket dispensers when you wait in line at a store.

This can be done by adjusting all of the **BankAccount** constructors so that they do not allow the user to "specify" the **accountNumber**. But



rather set it to the next available number and then increment the counter. Here is the code that we would need to write:

```
public class BankAccount {
    private static int LAST_ACCOUNT_NUMBER = 100000;

    private Customer    owner;
    private int         accountNumber;
    private float       balance;

    // This is the zero-parameter constructor
    public BankAccount() {
        owner = null;
        accountNumber = ++LAST_ACCOUNT_NUMBER;
        balance = 0;
    }

    // This is a 1-parameter constructor
    public BankAccount(Customer c) {
        owner = c;
        balance = 0;
        accountNumber = ++LAST_ACCOUNT_NUMBER;
    }

    // These are the get methods
    public Customer getOwner() { return owner; }
    public int     getAccountNumber() { return accountNumber; }
    public float   getBalance() { return balance; }
}
```

Notice that each bank account will always get a new number because all available constructors increment the global counter before assigning the bank account number to the new account. Also, notice that there are no **set** methods. That is because an account should never be allowed to change its owner nor its account number once it has been created. Also, there should not be any permission to directly modify (i.e., set) the balance ... there should be deposit and withdrawal procedures that must be followed.

Now, what about a **toString()** method? What should a bank account look like when printed? That is up to us. Perhaps we want it to look like this:

```
Bank Account #100001 with balance $1765.92
```

The above does not display the account owner. Here is how we would write the code:

```
public String toString() {
    return "Bank Account #" + accountNumber + " with balance $" +
        String.format("%.1.2f", balance);
}
```

Of course, we need a way of depositing and withdrawing money:

```
public void deposit(float amount) {
    balance += amount;
}
```

```
public boolean withdraw(float amount) {
    if (amount <= balance) {
        balance -= amount;
        return true;
    }
    return false;
}
```

Notice that the **withdraw()** method returns a **boolean** that will inform us as to whether or not there was enough money in the account. Both of these methods would need to be added to the account.

Here is a test program to see if it all works:

```
public class AccountTestProgram {
    public static void main(String args[]) {
        BankAccount    b1, b2, b3;

        b1 = new BankAccount(new Customer("Tim", "Foil",
            new Address("12", "Elm St.", "613-555-5555"));
        b2 = new BankAccount(new Customer("Dan", "Sing",
            new Address("1267A", "Oak St.", "613-555-5556"));
        b3 = new BankAccount(new Customer("Fran", "Tick",
            new Address("4761", "Pine Cres.", "613-555-5557"));

        b1.deposit(125);
        b2.deposit(3245.02f);
        b2.withdraw(1000);
        b3.withdraw(20);

        System.out.println(b1);
        System.out.println(b2);
        System.out.println(b3);
    }
}
```

Here is the expected output:

```
Bank Account #100001 with balance $125.00
Bank Account #100002 with balance $2,245.02
Bank Account #100003 with balance $0.00
```


Notice that the account numbers assigned are consecutive (i.e., 100001, 100002 and 100003). Of course, each time we run the program, the account numbers start over at 100001 again. If we wanted to ensure that our code assigned new numbers even when we restart the program, we would have to store the last account number counter in a file and then re-save the changed counter each time to the file. We will discuss file I/O later on in the course.

Finally, we need a way of keeping the accounts all together. We can do this by making a **Bank** class which keeps an array of **BankAccount** objects. Since we are using arrays, we will also want to define a fixed size for the array ... perhaps defined as a constant. Here is what we can do:

```
public class Bank {
    private static final int    ACCOUNT_CAPACITY = 100;

    private String              name;
    private BankAccount[]      accounts;
    private int                  numberOfAccounts;

    public Bank(String n) {
        name = n;
        numberOfAccounts = 0;
        accounts = new BankAccount[ACCOUNT_CAPACITY];
    }

    // These are the get methods (set methods are not allowed)
    public String getName() { return name; }
    public BankAccount[] getAccounts() { return accounts; }
    public int getNumberOfAccounts() { return numberOfAccounts; }

    // This returns a string representation of the bank
    public String toString() {
        return name + " with " + numberOfAccounts + " accounts";
    }
}
```

Of course, we need a way of opening accounts at the bank:

```
// Add an account to the bank
private void addAccount(BankAccount b){
    if (numberOfAccounts < ACCOUNT_CAPACITY)
        accounts[numberOfAccounts++] = b;
}
```

Notice that the method is private. That is because we don't want others passing in accounts that may be invalid or ones that belong to different banks. Instead, we will make a public method as a means of creating a new account, given a **Customer**:

```
// Open a bank account for this customer
public void openAccount(Customer c){
    addAccount(new BankAccount(c));
}
```

We will want to also probably allow depositing and withdrawals from the accounts based on an account number:

```
// Deposit an amount of money into account with given accountNumber
public boolean deposit(int accNum, float amount) {
    for (int i=0; i<numberOfAccounts; i++) {
        if (accounts[i].getAccountNumber() == accNum) {
            accounts[i].deposit(amount);
            return true;
        }
    }
    return false;
}
```

```
// Withdraw an amount of money from account with given accountNumber
public boolean withdraw(int accNum, float amount) {
    for (int i=0; i<numberOfAccounts; i++) {
        if (accounts[i].getAccountNumber() == accNum)
            return accounts[i].withdraw(amount);
    }
    return false;
}
```

We can then write any interesting methods that we want such as these:

```
// Determine total of all account balances
public float totalOfAllBalances() {
    float answer = 0;

    for (int i=0; i<numberOfAccounts; i++) {
        answer += accounts[i].getBalance();
    }
    return answer;
}
```

```
// List all accounts
public void listAccounts() {
    for (int i=0; i<numberOfAccounts; i++)
        System.out.println(accounts[i]);
}
```

We can test it all out using the following program:

```
public class BankTestProgram {
    public static void main(String[] args) {
        // Make a Bank
        Bank myBank = new Bank("Mark's Bank");

        // Make some bank accounts with customers
        myBank.openAccount(new Customer("Tim", "Foil",
            new Address("12", "Elm St.", "613-555-5555"));
        myBank.openAccount(new Customer("Dan", "Sing",
            new Address("1267A", "Oak St.", "613-555-5556"));
        myBank.openAccount(new Customer("Fran", "Tick",
            new Address("4761", "Pine Cres.", "613-555-5557"));

        myBank.deposit(100001, 125);
        myBank.deposit(100002, 3245.02f);
        myBank.withdraw(100002, 1000);
        myBank.withdraw(100003, 20);

        System.out.println("\nHere are the bank accounts:");
        myBank.listAccounts();

        System.out.println("\n\nThe bank has this much money: $" +
            String.format("%.1.2f", myBank.totalOfAllBalances()));
    }
}
```

Here is the expected output:

```
Here are the bank accounts:
Bank Account #100001 with balance $125.00
Bank Account #100002 with balance $2,245.02
Bank Account #100003 with balance $0.00

The bank has this much money: $2,370.02
```

As you can see, object-oriented programming requires you to define and implement many objects and to get them to work together in meaningful ways. Often, the object class definitions that you write can be re-used in many applications. It is therefore a good idea to ensure that these objects are robust and that their methods provide proper results. To do this, you should perform proper testing of your objects.



Unfortunately, testing is often tedious. It is therefore poorly done and ignored by many programmers. Companies that hire programmers do not like laziness ... and even worse ... they hate code with bugs or errors in it. To avoid disappointing your boss, possibly losing your job, and just to feel good about the quality of your work ... you should properly test your code.

Normally, it is not common to test your constructors nor get/set methods, but it is certainly important to test methods that perform computations, search, sort, etc... For problems that require numerical parameters, it is a good idea to test different values that could potentially cause problems. For example, if we were to fully test the **deposit()** method for the **BankAccount** class, we would want to test depositing the following amounts:

- 0.0 // deposit nothing
- 0.67 // a cents amount
- 100.57 // a typical positive amount
- 100.2234343 // an amount with many decimal places
- -34 // an invalid amount

We could create a simple test program to do this, making sure that we properly display the results to confirm that they are correct as follows ...

```
public class DepositTestProgram {
    public static void main(String args[]) {
        BankAccount acc;

        acc = new BankAccount(new Customer("Rusty", "Can",
            new Address("33", "Birch Ave."), "613-555-5558"));
        System.out.println("Account at start: " + acc);
        acc.deposit(0.0f);
        System.out.println("Account after depositing $0.00: " + acc);
        acc.deposit(0.67f);
        System.out.println("Account after depositing $0.67: " + acc);
        acc.deposit(100.57f);
        System.out.println("Account after depositing $100.57: " + acc);
        acc.deposit(100.2234343f);
        System.out.println("Account after depositing $100.2234343: " + acc);
        acc.deposit(-34);
        System.out.println("Account after depositing $-34: " + acc);
    }
}
```

Here is the output:

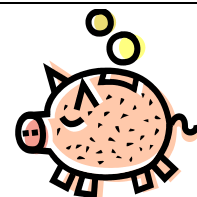
```
Account at start:                Account #100000 with $0.00
Account after depositing $0.00:   Account #100000 with $0.00
Account after depositing $0.67:   Account #100000 with $0.67
Account after depositing $100.57: Account #100000 with $101.24
Account after depositing $100.2234343: Account #100000 with $201.46
Account after depositing $-34:    Account #100000 with $167.46
```

Notice the careful use of **System.out.println()** in the program to provide a kind of “log” showing exactly what we tested and the order that things were tested in. If you were to read the output, you should be able to follow along as the deposit transactions were made to confirm the correct balance each time.

From the output, you may notice something that needs changing. For example, you may decide to prevent depositing negative amounts of money. You might do this by changing the code to generate an exception (more on this later) or perhaps simply perform a check and ignore deposits of negative amounts.

It really depends on the application and whether or not it is tied-in with the user interface. For example, at a bank machine, it is impossible to deposit a negative amount of money because the machine does not allow you to enter a negative sign. In such a situation, you may choose simply to ignore the problem altogether, since it would never occur. However, a simple check may be best, in case you port your code into a different program:

```
public void deposit(float amount) {
    if (amount > 0)
        balance += amount;
}
```



Then we would re-run the same test code to see whether or not it worked:

```
Account at start:                Account #100000 with $0.00
Account after depositing $0.00:   Account #100000 with $0.00
Account after depositing $0.67:   Account #100000 with $0.67
Account after depositing $100.57: Account #100000 with $101.24
Account after depositing $100.2234343: Account #100000 with $201.46
Account after depositing $-34:    Account #100000 with $201.46
```

Now this was a simple test program which is often known as a “Test Unit”. In larger, more complicated, real-world programs, in order to keep organized, it would be necessary to create multiple simple test units that test particular aspects of the program. For example,

```

public class BankAccountTestUnit1 {
    public static void main(String args[]) {
        BankAccount acc = new BankAccount(new Customer("Rusty", "Can",
            new Address("33", "Birch Ave."), "613-555-5558"));

        System.out.println("Account before depositing $100.57: " + acc);
        acc.deposit(100.57f);
        System.out.println("Account after depositing $100.57: " + acc);
    }
}

```

```

public class BankAccountTestUnit2 {
    public static void main(String args[]) {
        BankAccount acc = new BankAccount(new Customer("Rusty", "Can",
            new Address("33", "Birch Ave."), "613-555-5558"));

        System.out.println("Account before withdrawing $100: " + acc);
        acc.withdraw(100);
        System.out.println("Account after withdrawing $100: " + acc);
    }
}

```

In fact, it is often the case that we would like to perform transactions and test cases on a particular bank account. In this case, we can break down the separate test units as test methods in a larger test program:

```

public class BankAccountTestUnit3 {
    public static void deposit1(BankAccount acc) {
        System.out.println("Account before depositing $100.57: " + acc);
        acc.deposit(100.57f);
        System.out.println("Account after depositing $100.57: " + acc);
    }
    public static void deposit2(BankAccount acc) {
        System.out.println("Account before depositing $0.01: " + acc);
        acc.deposit(0.01f);
        System.out.println("Account after depositing $0.01: " + acc);
    }
    public static void withdraw1(BankAccount acc) {
        System.out.println("Account before withdrawing $100.57: " + acc);
        acc.withdraw(100.57f);
        System.out.println("Account after withdrawing $100.57: " + acc);
    }
    public static void withdraw2(BankAccount acc) {
        System.out.println("Account before withdrawing $0.01: " + acc);
        acc.withdraw(0.01f);
        System.out.println("Account after withdrawing $0.01: " + acc);
    }
}

```

```
public static void main(String args[]) {
    BankAccount acc;

    acc = new BankAccount(new Customer("Rusty", "Can",
                                       new Address("33", "Birch Ave.", "613-555-5558"));
    acc.deposit(0);

    deposit1(acc);
    deposit2(acc);
    withdraw1(acc);
    withdraw2(acc);

    acc = new BankAccount(new Customer("Ann", "Tenna",
                                       new Address("84", "Maple Ave.", "613-555-5559"));
    acc.deposit(10);

    deposit1(acc);
    deposit2(acc);
    withdraw1(acc);
    withdraw2(acc);

    acc = new BankAccount(new Customer("Ella", "Vator",
                                       new Address("873", "Spruce Dr.", "613-555-5560"));
    acc.deposit(200);

    deposit1(acc);
    deposit2(acc);
    withdraw1(acc);
    withdraw2(acc);
}
}
```

There are actually principles and guidelines for writing test cases for large systems. However, it is beyond the scope of this course. You will learn more about proper testing next year.

This page was intentionally left blank.

Chapter 4

Class Hierarchies and Inheritance

What is in This Chapter ?

This chapter discusses how objects are organized into a **class hierarchy** and then explains the notion of **inheritance** as a means of sharing attributes and behaviors among classes. It also explains the notion of **abstract classes** and java **interfaces** that allow seemingly unrelated classes to share common behavior.



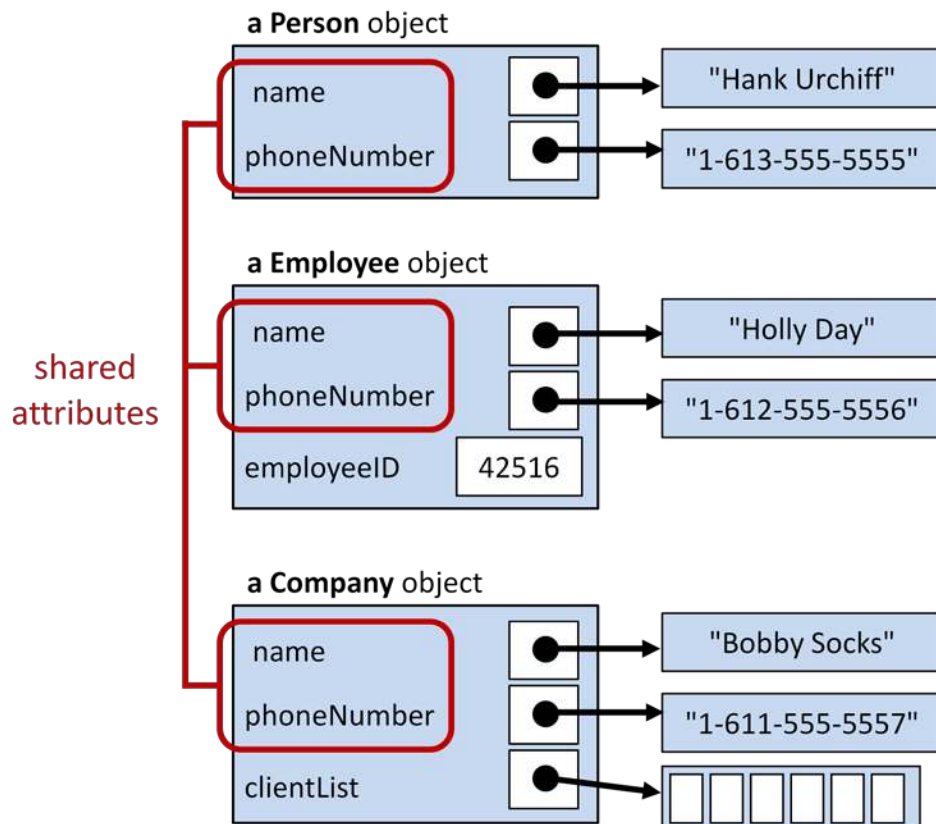
4.1 Organizing Classes

As we have already seen, defining objects as a new kind of data structure simply involves created new **classes**, each in their own file (e.g., Car.java, Person.java, Address.java, House.java, etc.). In fact, a definition of the word 'class' in English is:

"A collection of things sharing a common attribute".

So, for example, when we create a **Person** class, we are implying that all **Person** objects have some attributes in common. Similarly, a **Car** class would define the common attributes that all **Car** objects have. In general, since **Person** and **Car** are different classes, their list of attributes will differ.

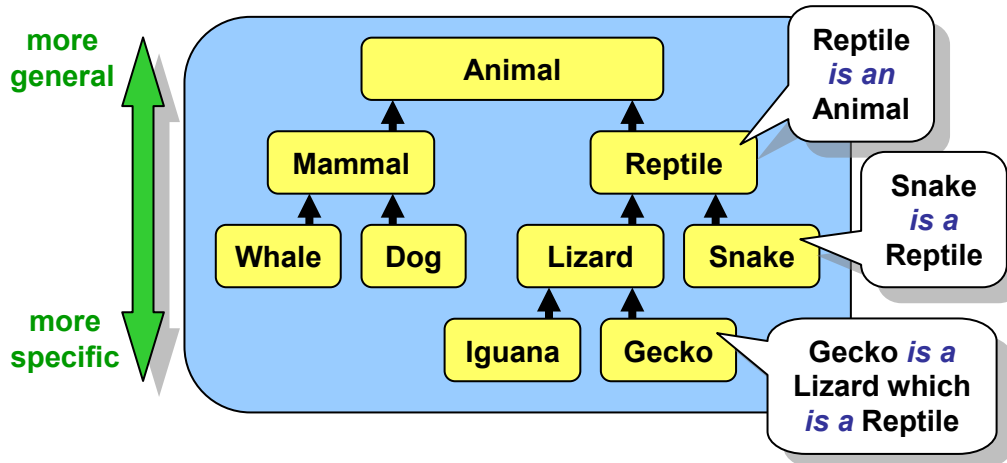
In real life, however, there are some objects that "share" attributes in common. For example, **Person** objects may have **name** and **phoneNumber** attributes, but so can **Employee**, **Manager**, **Customer** and **Company** objects. Yet, there may be additional attributes of these other objects that **Person** does not have. For example, an **Employee** object may maintain **employeeID** information or a **Company** object may have a **clientList** attribute, whereas **Person** objects in general do not keep such information:



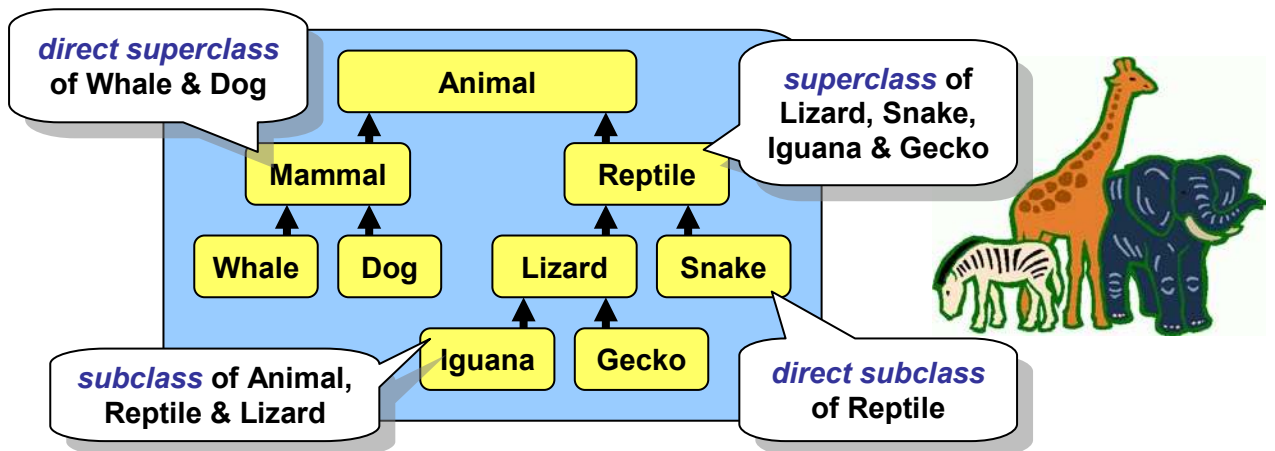
In addition to commonality between attributes, classes may also share common behavior. That is, two or more objects may have the ability to perform the same function or procedure. For example, if a **Person**, **Car** and **Company** are all *insurable*, then they may all have a function called **calculateInsurancePremium()** that determines the pricing information for their insurance plan.

All object-oriented languages (e.g., JAVA) allow you to organize your classes in a way that allows you to take advantage of the commonality between classes. That is, we can define a class with certain attributes (and/or behaviors) and then specify which other classes share those same attributes (and/or behaviors). As a result, we can greatly reduce the amount of duplicate code that we would be writing by not having to re-define the common attributes and/or behaviors for all of the classes that share such common features.

JAVA accomplishes this task by arranging all of its classes in a "family-tree"-like ordering called a **class hierarchy**. A class hierarchy is often represented as an upside down tree (i.e., the root of the tree at the top). The more "general" kinds of objects are higher up the tree and the more "specific" (or specialized) kinds of objects are below them in the hierarchy. So, a **child** object defined in the tree is a *more specific kind* of object than its **parent** or **ancestors** in the tree. Hence, there is an "**is a**" (i.e., "is-a-kind-of") relationship between classes:



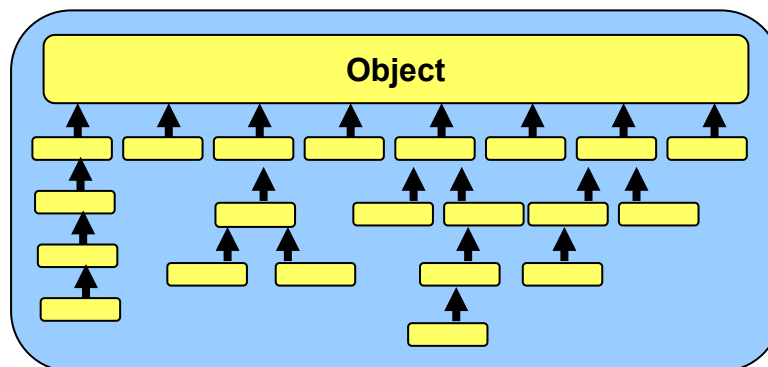
Each class is a **subclass** (i.e., a specialization) of some other class which is called its **superclass** (i.e., a generalization). The **direct superclass** is the class right "above" it:



Here, **Snake**, and **Lizard** are subclasses of **Reptile** (i.e., they are special kinds of reptiles). Also **Whale** and **Dog** are subclasses of **Mammal**. All of the classes are subclasses of **Animal** (except **Animal** itself). **Animal** is a superclass of all the classes below it, and **Mammal** is a

superclass of **Whale** and **Dog**. As we can see, we can go even deeper in the hierarchy by creating subclasses of **Lizard**. Usually, when we use the term *superclass*, we are referring to the class that is directly above a particular class (i.e., the direct superclass).

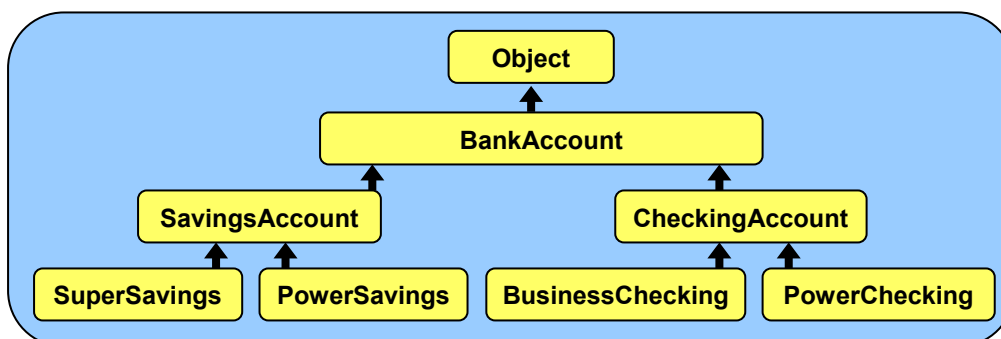
The **Animal** hierarchy above represents a set of classes that we may define ourselves. But where do they fit-in with all the other pre-made JAVA classes like **String**, **Date**, **Rectangle** etc... ? Well, all objects have one thing in common ... they are all *Objects*. Hence, at the very top of the hierarchy is a class called **Object**. Therefore, all classes in JAVA are *subclasses* of **Object**:



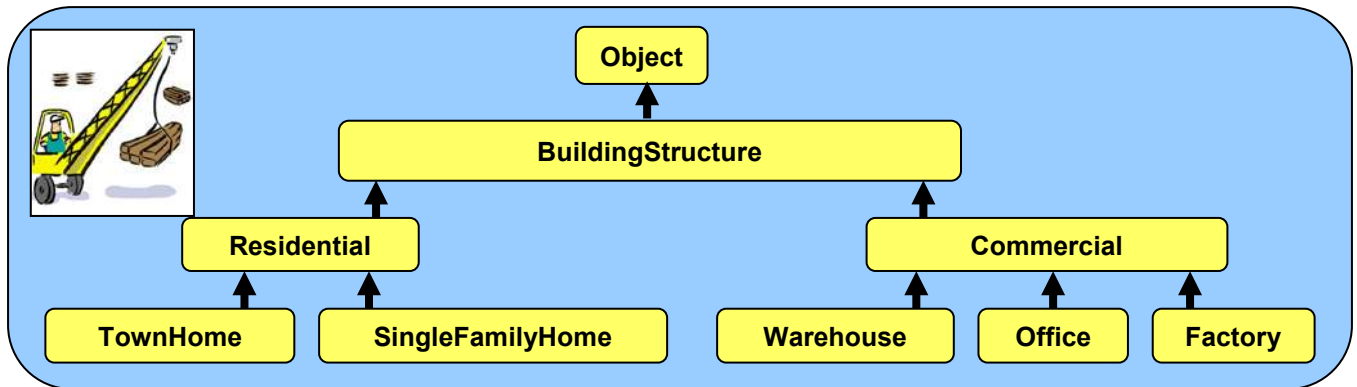
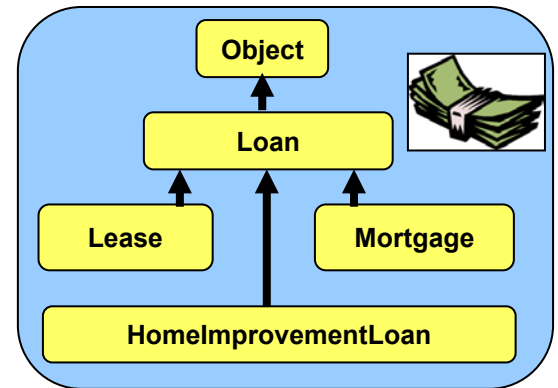
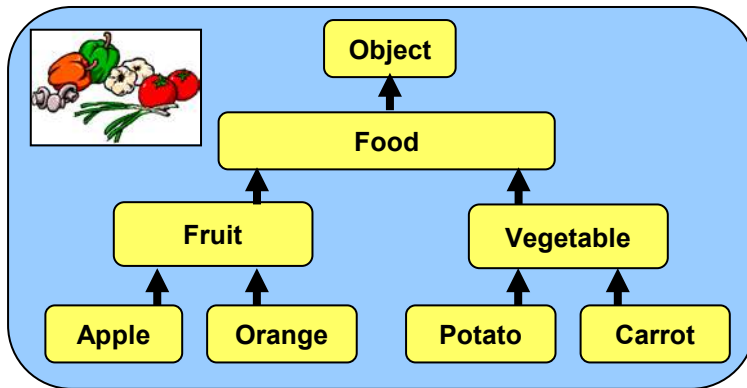
All of the classes that we created so far have been *direct* subclasses of **Object**. That means that they did not share attributes with one another, but that they shared attributes only with **Object**. However, we have the freedom to re-arrange our classes in a manner that will allow them to share attributes with one other.

The way in which we arrange our classes will depend on *how similar* our objects are with respect to their attributes. For example, a **Car** and a **Truck** have something in common ... they are both *drivable*. Whereas an **MP3Player** and a **BankAccount** have little or nothing in common with **Car** or **Truck** objects. So, intuitively, **Car** and **Truck** classes should somehow be grouped together (i.e., placed nearby) in the hierarchy.

As an example, consider creating many kinds of bank accounts. We might arrange them in a hierarchy like this:



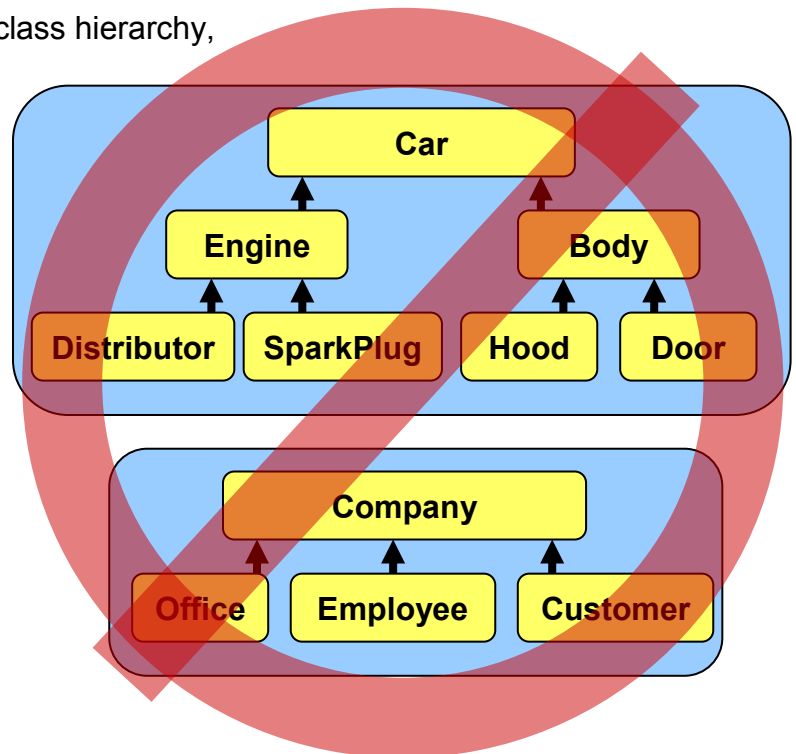
Here are a few more examples of hierarchies of classes that we may create:



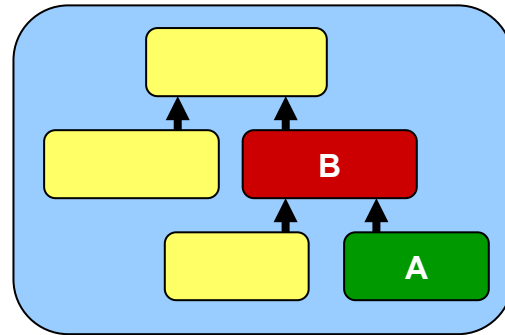
We will talk more about **how** and **why** we arrange these classes as above. But remember, a class should only be a subclass of another class if it "is a kind of" its superclass.

Sometimes, students misunderstand the class hierarchy, thinking that a class becomes a subclass of another one if the superclass "is made of" the subclasses.

That is, they mistakenly assume that it is a "has a" relationship instead of an "is a" relationship. Therefore, the following hierarchies would be wrong →



In JAVA, in order to create a subclass of another class, use the **extends** keyword in our class definition. For example, assume that we wanted to ensure that class **A** was placed in the hierarchy as a subclass of class **B** as shown here.



To make this happen, we simply write **extends B** immediately after we specify name of class **A** as follows:

```

public class A extends B {
    ...
}
    
```

If the **extends** keyword is not used (i.e., as we left it out from all our previous class definitions), it is assumed that the class being defined extends the **Object** class. So, all the classes that we defined previously were direct subclasses of **Object**.

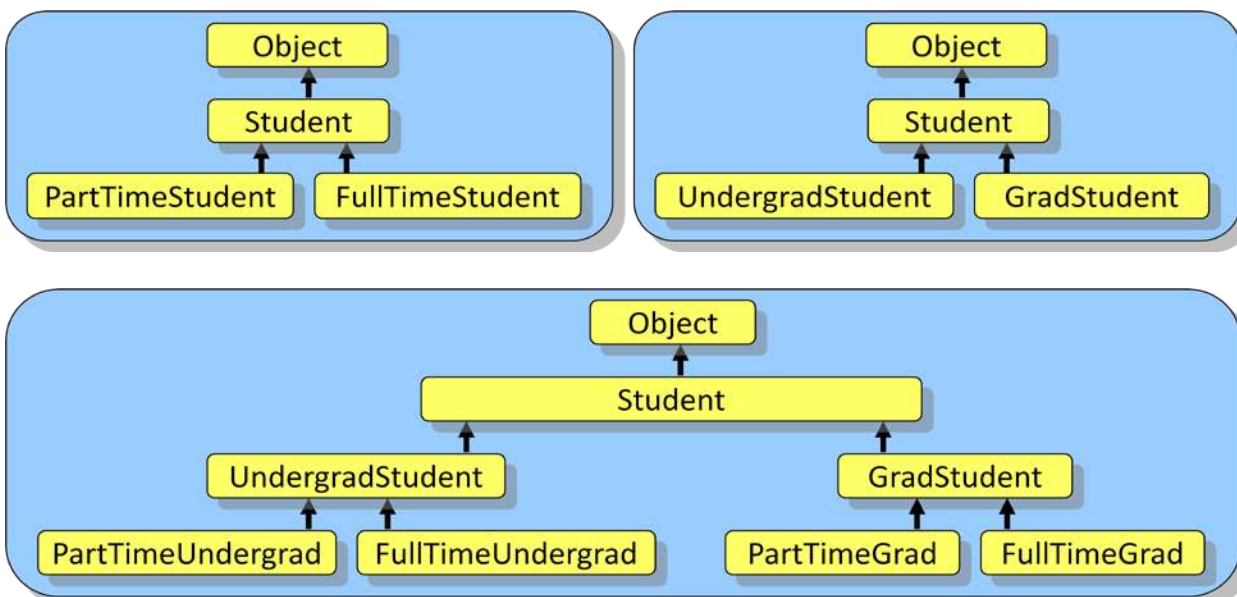
How do we know how deep we should make the class hierarchy (i.e., tree) ?

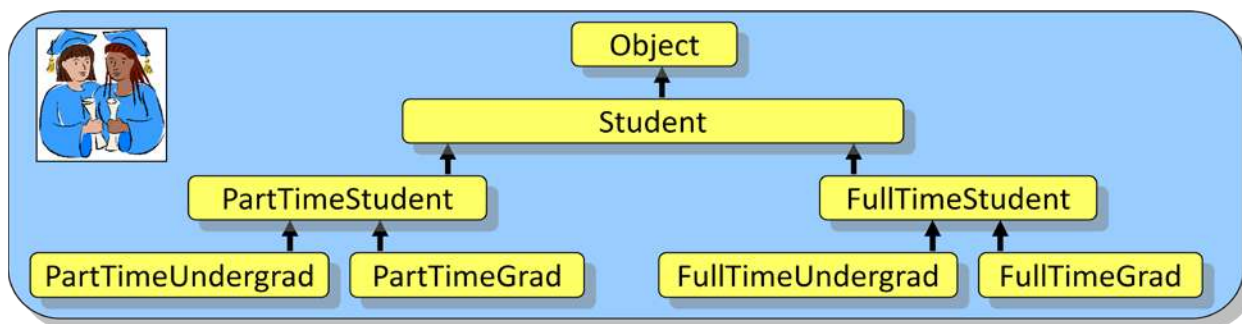
Most of the time, any “**is a**” relationship between objects should certainly result in subclassing. Object-oriented code usually involves a lot of small classes as opposed to a few large ones.



It is often the case that our class hierarchies become rearranged over time, because we often make mistakes in deciding where to place the classes. We make such mistakes because it is not always easy to choose a hierarchy ... it depends on the application.

For example, hierarchies of classes representing students in a university may be arranged in many different ways ... here are just 3 possibilities ...





How do we know which one to use ? It will depend on the state (i.e., attributes) and behavior (i.e., methods) that is common between the subclasses. If we find that the main differences in attributes or behavior are between full time and part time students (e.g., fee payment rules), then we may choose the top hierarchy. If however the main differences are between graduate and undergraduate (e.g., privileges, requirements, exam styles etc..), then we may choose the middle hierarchy. The bottom hierarchy further distinguishes between full and part time graduate and undergraduate students, if that needs to be done. So ... the answer is ... we often **do not know** which hierarchy to choose until we thought about which hierarchy allows the maximum sharing of code.

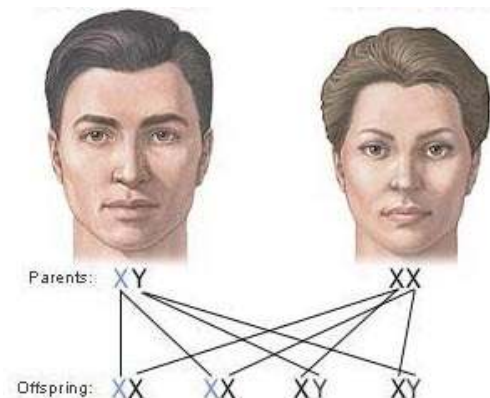
4.2 Inheritance

You may have heard the term *inherit* before which has various meanings in English such as:

- “to receive from a predecessor” or
- “to receive by genetic transmission”

Through birth, all of us have *inherited traits* and **behaviors** from our parents. Something similar happens in JAVA with regards to the class hierarchy. A subclass (i.e., child) *inherits* the attributes (i.e., instance variables) and behavior (i.e., methods) from all of its superclasses (i.e., ancestors in the class hierarchy). So as a general definition, in Object-Oriented Programming:

Inheritance is the act of receiving shared attributes and behavior from more general types of objects up the hierarchy.



This means that a subclass has the same "general" attributes/behaviors as its superclasses as well as possibly some new additional attributes/behaviors which are specific for the subclass. There are many advantages of using **Inheritance**:

- allows code to be **shared** between classes
... promotes software re-usability
- **saves programming time** since code is shared
...less code needs to be written
- helps keep **code simple** since inheritance is natural in real life



Some languages (e.g., C++) allow **Multiple Inheritance**, which means that a class can inherit state and behavior from more than one class. However, JAVA does not support multiple inheritance. We can however, partially "fake" it (with respect to methods) through the use of *interfaces* (which we will discuss later).

Consider making an object to represent an **Employee** in a company which maintains: **name**, **address**, **phoneNumber**, **employeeNumber** and **hourlyPay**. We may make a single class:

```
public class Employee {
    String    name;
    Address   address;
    String    phoneNumber;
    int       employeeNumber;
    float     hourlyPay;
    ...
}
```



Employee

Assume now that we have many employees in a company in which a few of them are managers. If the managers are all essentially the same as employees, except perhaps that they have a higher **hourlyPay**, then there is no need to create any new classes. The **Employee** class is sufficient to represent them.

However, what if there were some more significant differences between managers and employees? Perhaps it would be beneficial to create a separate class for them. We would need to determine **what is different** between these two classes with respect to their attributes and behaviors. For example, a **Manager** may have:

- *additional* attributes (e.g., a list of **duties**, a list of **employees** that work for them, etc...)
- *additional* (or different) behavior (e.g., they may compute their pay differently, or have different benefit packages, etc...)

In these situations, a **Manager** may be considered as a special "kind of" **Employee**. It would therefore make sense for the **Manager** to be a **subclass** of **Employee** as follows:

```
public class Employee {
    String    name;
    Address   address;
    String    phoneNumber;
    int       employeeNumber;
    float     hourlyPay;
    ...
}

public class Manager extends Employee {
    String[]  duties;
    Employee[] subordinates;
    ...
}
```



Employee

Manager

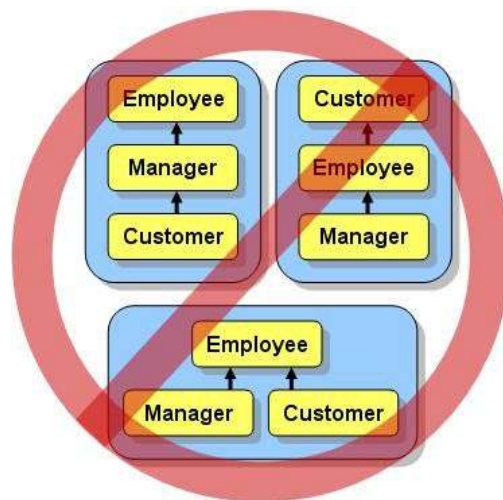
Notice here that **Manager** would inherit all of the attributes of the **Employee** class, so that **Employees** have 5 attributes, while **Managers** have 7. All **Employee behaviors** would also be inherited by **Managers**.

Now, what if we wanted to represent a **Customer** as well in our application? Our application may require keeping track of a customer's **name**, **address** and **phoneNumber**. But these attributes are also being used for our **Employee** objects. We could make two separate unrelated classes ... one called **Customer** ... the other called **Employee**. We could define **Customer** as follows:

```
public class Customer {
    String    name;
    Address   address;
    String    phoneNumber;
    ...
}
```



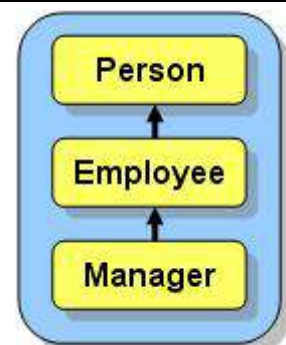
This would work fine. However, you will notice that both **Employee** and **Customer** have some attributes in common. So, if we defined the **Customer** class in this manner, we would need to repeat the same definitions, and perhaps some of the behaviors. It would be better if we could somehow use inheritance to allow **Customers** to share attributes and behaviors that are in common with **Employees**. So, we should perhaps have **Customer** inherit from something. We have a few choices. We can have **Customer** inherit from **Manager**, **Employee** inherit from **Customer** or **Customer** inherit from **Employee** as follows ...



However, neither of these hierarchies will work according to the "is a" relationship because (1) a **Customer** is not always a **Manager**, (2) an **Employee** is not always a **Customer**, and (3) a **Customer** is not always an **Employee**.

One possible solution is to change the name **Customer** to **Person**. In this way, a customer is simply represented by a **Person** object and we can use the following hierarchy:

```
public class Person {
    String    name;
    Address   address;
    String    phoneNumber;
}
public class Employee extends Person {
    int       employeeNumber;
    float     hourlyPay;
}
public class Manager extends Employee {
    String[]  duties;
    Employee[] subordinates;
}
```



Now **Employee** inherits 3 attributes from **Person**, so it has 5 altogether, while **Manager** inherits 3 from **Person** and 2 from **Employee**, making 7 altogether. **Customers**, are then represented simply as **Person** objects.

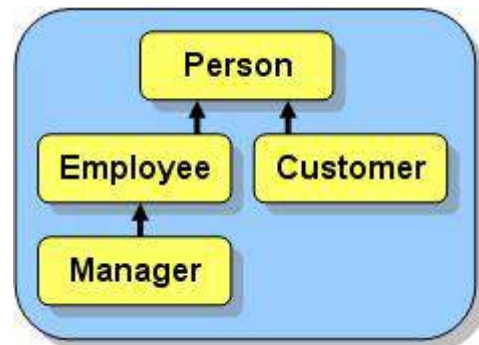
This is a good solution as long as ALL of the attributes (e.g., **name**, **address**, **phone number**) for a customer (i.e., **Person** object) is also shared with **Employee** and **Manager**. Also, there must not be any attributes or behaviors in the **Person** class that do not apply to an **Employee** and a **Manager**. For example, if the application required us to keep track of a list of items purchased by the customer or perhaps even a purchase history, then such attributes may not make sense for an **Employee** or **Manager**. So, if there is different behavior or attributes that is unique to customers, then we must create a separate **Customer** class to define these differences. In this case, we can still share the **name**, **address** and **phoneNumber** by creating an *extra* **Person** class to hold the common attributes. We can create the following hierarchy:

```
public class Person {
    String    name;
    Address   address;
    String    phoneNumber;
}

public class Employee extends Person {
    int       employeeNumber;
    float     hourlyPay;
}

public class Customer extends Person {
    String[]  itemsPurchased;
    Date[]    purchaseHistory;
}

public class Manager extends Employee {
    String[]  duties;
    Employee[] subordinates;
}
```



This will allow all common attributes (i.e., **name**, **address**, **phoneNumber**) to be shared by all the classes while allowing **Customer** objects to have their own attributes and behaviors.

At this point, we should clarify the advantages of the attribute-related inheritance that is occurring within our hierarchy. Here is a simple example piece of code showing the attributes that are readily available to each type of object defined in our example ...

```

Person    p = new Person();
Employee  e = new Employee();
Customer  c = new Customer();
Manager   m = new Manager();

p.name = "Hank Urchiff";           // own attribute
p.address = new Address();         // own attribute
p.phoneNumber = "1-613-555-2328"; // own attribute

e.name = "Minnie Mumwage";        // attribute inherited from Person
e.address = new Address();        // attribute inherited from Person
e.phoneNumber = "1-613-555-1231"; // attribute inherited from Person
e.employeeNumber = 232867;        // own attribute
e.hourlyPay = 8.75f;              // own attribute

c.name = "Jim Clothes";           // attribute inherited from Person
c.address = new Address();        // attribute inherited from Person
c.phoneNumber = "1-613-555-5675"; // attribute inherited from Person
c.itemsPurchased[0] = "Pencil Case"; // own attribute
c.purchaseHistory[0] = Date.today(); // own attribute

m.name = "Max E. Mumwage";        // attribute inherited from Person
m.address = new Address();        // attribute inherited from Person
m.phoneNumber = "1-613-555-8732"; // attribute inherited from Person
m.employeeNumber = 232867;        // attribute inherited from Employee
m.hourlyPay = 8.75f;              // attribute inherited from Employee
m.duties[0] = "Phone Clients";    // own attribute
m.subordinates[0] = e;           // own attribute

```

Notice that we use the inherited attributes just as if they were defined as part of that class directly. For example, the **Employee** object **e**, **Customer** object **c** and **Manager** object **m**, all access the **name** attribute as if it was defined in their class ... even though it is actually defined in the **Person** class ... written in a different **.java** file!! You can see that through inheritance, we do not have to re-define the **name** attribute in each of these classes. The same holds true for the **address** and **phoneNumber** attributes, as well as any other inherited attributes in the subclasses.

At this point, we only examined how to decide upon a class hierarchy based on the differences in attributes. However, we would have to think in the same manner by examining the behaviors of the individual classes. For example, even if managers did not have the **duties** and **subordinates** attributes shown above, we may still want to make a separate class for managers if there are behaviors that differ (e.g., different **computePay()** method).

Now, we will consider an example that shows how inheritance applies to behaviors within a simple hierarchy of **BankAccount** objects.

Consider creating an application for a bank that maintains account information for its customers. All bank accounts at this bank must maintain 3 common attributes (the **owner's** name, the **account number** and the **balance** of money remaining in the account).

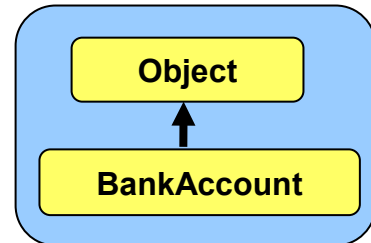
Also, an account, by default, should have simple behaviors to **deposit** and **withdraw** from the account. So, in its simplest form, a **BankAccount** object can be defined and used as follows:

```
public class BankAccount {
    String    owner;           // person who owns the account
    int      accountNumber;   // the account number
    float    balance;        // amount of money currently in the account

    // Some constructors
    public BankAccount() {
        this.owner = "";
        this.accountNumber = 0;
        this.balance = 0;
    }
    public BankAccount(String ownerName) {
        this.owner = ownerName;
        this.accountNumber = 0;
        this.balance = 0;
    }

    // Deposit money into the account
    public void deposit(float amount) {
        this.balance += amount;
    }

    // Withdraw money from the account
    public void withdraw(float amount) {
        if (this.balance >= amount)
            this.balance -= amount;
    }
}
```



Now assume that the bank wants to distinguish between “*savings*” accounts and “*non-savings*” accounts in that the customer cannot withdraw money from a “*savings*” account once it has been deposited (i.e., to get the money out of the account, the customer must close the account).



We would need to have a way of disabling the withdraw behavior for *savings* accounts. We could do this through inheritance by creating a subclass of **BankAccount** to represent a special “kind of” account ... we will call it **SavingsAccount**:

```
public class SavingsAccount extends BankAccount {
}
```

Just by writing this simple “virtually empty” class definition in which **SavingsAccount** *extends* **BankAccount**, we have “invented” a new type of bank account that inherits all 3 attributes from **BankAccount** as well as the **deposit()** and **withdraw()** methods.



We could verify this by writing a simple piece of test code:

```
SavingsAccount s = new SavingsAccount();

System.out.println(s.balance);           // displays 0.0

s.deposit(120);
System.out.println(s.balance);           // displays 120.0

s.withdraw(20);
System.out.println(s.balance);           // displays 100.0
```

Something important to know, however, is that a subclass does not automatically inherit the constructors in its superclass. So, **SavingsAccount** does not inherit the two constructors in **BankAccount** ... but it does get to use its own default constructor (i.e., zero-parameter constructor) for free. We can verify this by altering the first line in our test code so read:

```
SavingsAccount s = new SavingsAccount("Bob");
```

If we made such an alteration to the code, our test code would no longer compile. We would receive the following compile error:

```
cannot find symbol constructor SavingsAccount(java.lang.String)
```

which is telling us that we don't have a constructor in our **SavingsAccount** class that takes a single **String** parameter. How then did our `new SavingsAccount()` code work previously since it seems to have properly initialized the account number? Well, as it turns out, the default constructor that we get for free actually looks as follows:

```
public SavingsAccount() {
    super();
}
```

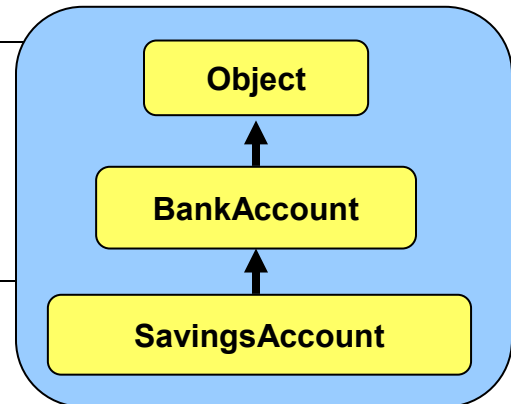
What does this mean? What does the keyword **super** do? The keyword **super** is actually a special word that represents the *superclass* of this class. In our case, the **super** class is **BankAccount**. So, it is essentially doing a call to **BankAccount()** ... which means it is calling the superclass constructor.

Therefore, if we want to make use of the attribute initialization code that is in a constructor in a superclass, we can call the superclass constructor from our own by using **super(...)** along with the appropriate parameters.



Hence we can write the following constructor in our **SavingsAccount** class:


```
public class SavingsAccount extends BankAccount {
    public SavingsAccount(String aName) {
        super(aName);
    }
}
```



If we do this, then we can use the following code without compile errors:

```
SavingsAccount s = new SavingsAccount("Bob");
```

Keep in mind, however, that the list of parameters (i.e., the types) supplied within the **super(...)** call, must match the list of parameters (i.e., the types) of one of the constructors in the superclass. In order to see the advantage of using constructor inheritance, here is what the code would look like with and without using inherited constructors:

Without Inheritance (need to re-write code)	With Inheritance
<pre>public SavingsAccount() { { this.owner = ""; this.accountNumber = 0; this.balance = 0; } public SavingsAccount(String ownerName) { this.owner = ownerName; this.accountNumber = 0; this.balance = 0; }</pre>	<pre>public SavingsAccount() { super(""); } public SavingsAccount(String aName) { super(aName); }</pre> 

Again ... the amount of code that needs to be written is reduced when using inheritance. So, we have **SavingsAccount** properly inheriting from **BankAccount**, however, the **SavingsAccount** class still allows withdrawals. In order to disable this behavior, we need to somehow “prevent” the withdraw method code from being used by savings accounts. The simplest and most common way of doing this is to write a new **withdraw()** method in the **SavingsAccount** class that simply does nothing as follows ...

```
public class SavingsAccount extends BankAccount {
    // Constructor to call the superclass constructor
    public SavingsAccount(String aName) { super(aName); }
    public SavingsAccount() { super(""); }

    // Prevent the withdrawal of money from the account
    public void withdraw(float amount) {
        // Do nothing
    }
}
```

Once we re-compile, we can test it out by running our test code again:

```
SavingsAccount s = new SavingsAccount();

System.out.println(s.balance);           // displays 0.0

s.deposit(120);
System.out.println(s.balance);           // displays 120.0

s.withdraw(20); // this will do nothing now
System.out.println(s.balance);           // displays 120.0
```

Notice that the test code remains the same but now it no longer performs the withdrawal calculation. What is actually happening here? By writing the `withdraw()` method in the **SavingsAccount** class, we are actually *overriding* the one that is in the **BankAccount** class. That is, we are replacing the inherited behavior with our own unique behavior. So, we are *preventing* or *disabling* the inheritance for this behavior.

At this point, we now have **SavingsAccounts** that cannot be withdrawn from and normal **BankAccounts** that can be withdrawn from. Let us see another way that we can use overriding ... to *modify* inherited behavior.

Assume that the bank also wants to encourage depositing to savings accounts by giving \$0.50 to the customer for each \$100 that they deposit into their **SavingsAccount** (i.e., not for regular **BankAccounts**). For example, if they deposit \$354.23, then their account balance should immediately increase by \$354.73 ... showing the extra \$0.50 applied to the deposit amount.



To do this, we can completely override the deposit method from **BankAccount** by writing the following method in **SavingsAccount** ...

```
// Deposit money into the account
public void deposit(float amount) {
    this.balance += amount;

    // Now add the bonus 50 cents per $100
    int wholeDollars = (int)(amount/100);
    this.balance += wholeDollars * 0.50f;
}
```

This method of overriding would work fine and would properly add the extra bonus deposit incentive. However, the first line is a duplication of the **BankAccount** class's `deposit()` method. This duplication may seem insignificant in this simple example, but in a real bank application there may actually be much more code devoted to the deposit process (e.g., logging the transaction). Hence, it would be better to make use of inheritance.

How though, can we inherit the **deposit()** method in **BankAccount**, while also incorporating the additional bonus deposit behavior necessary for **SavingsAccounts** ? The answer makes use of the **super** keyword again. Here is the solution:

```
// Deposit money into the account
public void deposit(float amount) {
    // Call the deposit() method in the superclass
    super.deposit(amount);

    // Now add the bonus 50 cents per $100
    int wholeDollars = (int)(amount/100);
    this.balance += wholeDollars * 0.50f;
}
```



Notice that this time we use a dot **.** after the **super** keyword, followed by the method that we want to call in the superclass. Here, the word **super** is used to tell JAVA to look for the **deposit()** method in the superclass. JAVA will go and evaluate the superclass **deposit()** method (which performs the “normal” depositing process) and then return here and complete the behavior by adding the 50 cent bonus incentive. This method is still considered to *override* the **deposit()** method in **BankAccount**. It is an example of a situation in which we want to “borrow” a superclass’s behavior, but then add some additional behavior as well.

Alternatively, we could have combined the deposit amount with the 50 cent bonus incentive before calling the superclass method as follows:

```
// Deposit money into the account
public void deposit(float amount) {
    int wholeDollars = (int)(amount/100);
    super.deposit(amount + (wholeDollars * 0.50f));
}
```

or even simpler:

```
// Deposit money into the account
public void deposit(float amount) {
    super.deposit(amount + (int)(amount/100)* 0.50f);
}
```



I’m sure you will agree that the overriding can be quite powerful tool to save coding time.

Just so you understand ... what would happen if we used **this** instead of **super** as follows:


```
// Deposit money into the account
public void deposit(float amount) {
    this.deposit(amount + (int)(amount/100)* 0.50f);
}
```

Well, we would be asking JAVA to call the **deposit()** method in *this* class, not the one in **BankAccount**. Furthermore, since this code is written *inside* the **deposit()** method, we are telling JAVA to call the method that we are actually trying to write! So the method will keep calling itself forever ... an infinite loop! We would get a pile of runtime error messages that says something like this:

```
Exception in thread "main" java.lang.StackOverflowError
  at SavingsAccount.deposit(SavingsAccount.java:13)
  at SavingsAccount.deposit(SavingsAccount.java:13)
  at SavingsAccount.deposit(SavingsAccount.java:13)
  ...
  at SavingsAccount.deposit(SavingsAccount.java:13)
```



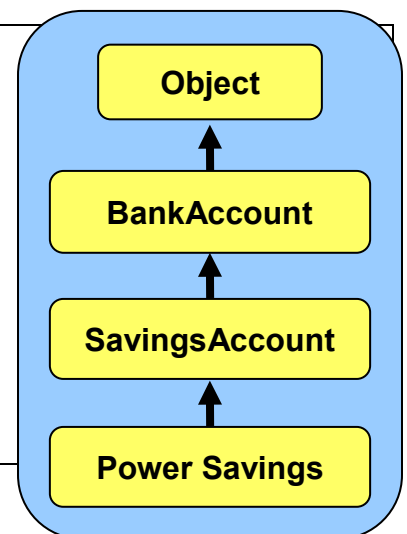
OK. Now assume that the bank application needs to further distinguish between accounts in that it also has a special “power savings” account that is a special type of savings account that allows withdrawals, but there is a \$1.25 service fee each time a withdrawal is made. As before, this new type of account should also have the 50 cent incentive for each \$100 deposited.



Assuming that we call the new class **PowerSavings**, where do we put it in the hierarchy? We need it to inherit the **deposit()** method from **SavingsAccount** but the **withdraw()** method from **BankAccount**. If we make **PowerSavings** a subclass of **SavingsAccount**, we will inherit the **deposit()** behavior that we want, but would then need to write a new **withdraw()** method, since the one in **SavingsAccount** does nothing. We could do this ...

```
public class PowerSavings extends SavingsAccount {
    // Constructor to call the superclass constructor
    public PowerSavings(String aName) {super(aName);}
    public PowerSavings() {super("");}

    // Withdraw money from the account
    public void withdraw(float amount) {
        if (this.balance >= (amount + 1.25f))
            this.balance -= (amount + 1.25f);
    }
}
```



This code would work fine.

Again, we are using *overriding* by having the **withdraw()** method in **PowerSavings** override the default behavior in **SavingsAccount**. We can test our new class with the following test code:

```
PowerSavings    s = new PowerSavings();

System.out.println(s.balance);           // displays 0.0

s.deposit(320);
System.out.println(s.balance);           // displays 321.50

s.withdraw(20);
System.out.println(s.balance);           // displays 300.25
```

Notice that the **withdraw()** method properly deducts the \$1.25 fee.

However, again we are duplicating code. The code here is small, however in a large system, there may be more complicated code for withdrawing money (e.g., transaction logging, overdraft allowances, etc...). So, we do not want to duplicate this code. In fact, it would be nice if we could do something like this to call the **withdraw()** method code up in **BankAccount**:

```
public class PowerSavings extends SavingsAccount {
    // Constructor to call the superclass constructor
    public PowerSavings(String aName) { super(aName); }
    public PowerSavings() { super(""); }

    // Withdraw money from the account
    public void withdraw(float amount) {
        super.withdraw(amount + 1.50f);
    }
}
```

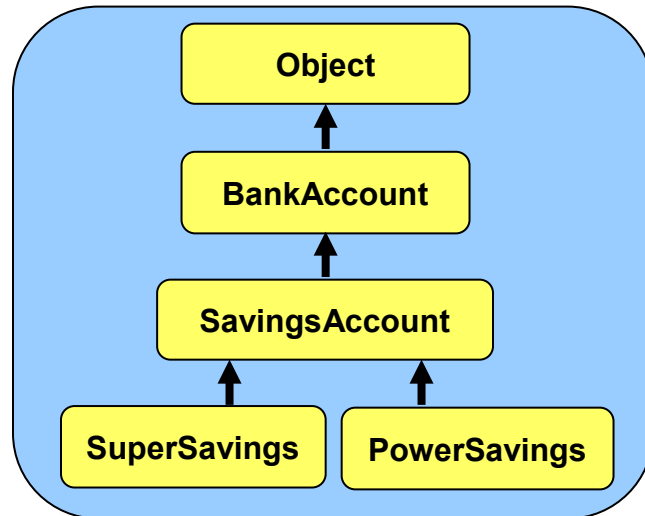
But this won't work. Why not? Because **super** refers to the **SavingsAccount** class here, and so it calls the **withdraw()** method in **SavingsAccount** that does nothing. In a way, what we want to do is something like this:

```
super.super.withdraw(amount + 1.50f); // super-duper does not work
```



Unfortunately, we cannot skip over a class when looking up the class hierarchy for a method. What can we do then? The solution is to re-organize our hierarchy. We seem to need common deposit behavior for savings accounts, but then differing withdrawal behavior. In reality, we actually need to distinguish between the two kinds of savings accounts. We will rename **SavingsAccount** to **SuperSavings** which will represent the previous savings account behavior. Then we will create a new **SavingsAccount** class that will contain the shared deposit behavior between the two types of savings accounts.

Here is the new hierarchy:



Here is the code:

```
public class SavingsAccount extends BankAccount {
    public SavingsAccount(String aName) { super(aName); }
    public SavingsAccount() { super(""); }

    public void deposit(float amount) {
        super.deposit(amount + (int)(amount/100)* 0.50f);
    }
}
```

```
public class SuperSavings extends SavingsAccount {
    public SuperSavings(String aName) { super(aName); }
    public SuperSavings() { super(""); }

    public void withdraw(float amount) { /* Do nothing */ }
}
```

```
public class PowerSavings extends SavingsAccount {
    public PowerSavings(String aName) { super(aName); }
    public PowerSavings() { super(""); }

    public void withdraw(float amount) {
        super.withdraw(amount + 1.50f);
    }
}
```

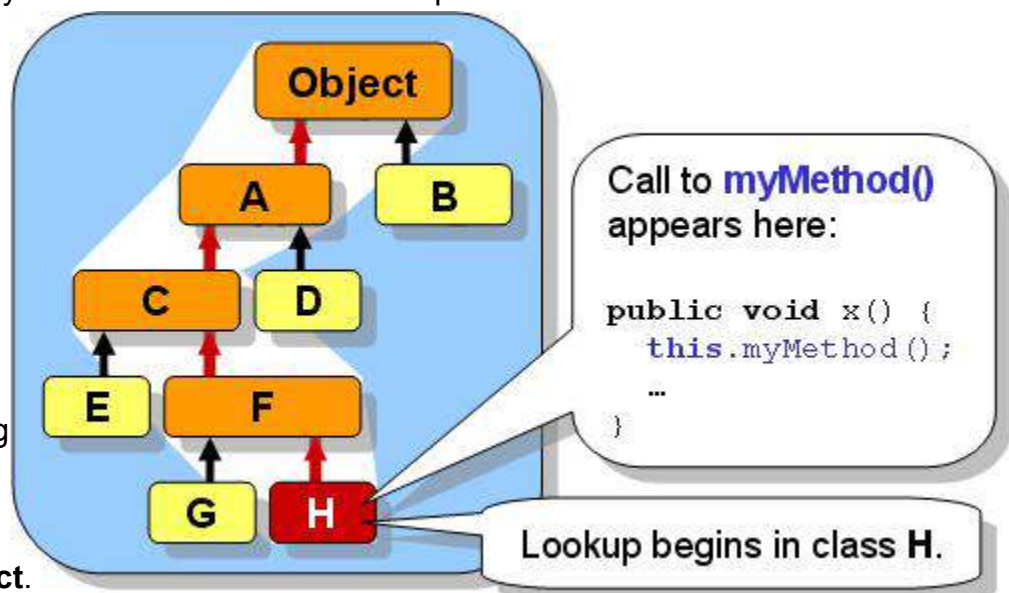
The code will work as we expect it to now, taking full advantage of inheritance.

To properly understand method calling and overriding when dealing with class hierarchies, we need to consider how JAVA "finds" a method in the class hierarchy when you try to call it. It can be confusing if there are many "overridden" methods (i.e., all with the same name and parameter lists), because we may not know which one JAVA will use. Fortunately, there is a simple way to figure this out.

Whenever you call a method from a class directly (e.g., **this.myMethod()**), JAVA looks first to see whether or not you have such a method in the class that you are calling it from. If it finds it there, it evaluates the code in that method. Otherwise, JAVA tries to look for the method up the hierarchy (never down the hierarchy) by checking the superclass. If not found there, JAVA continues looking up the hierarchy until it either finds the method that you are trying to call, or until it reaches the **Object** class at the top of the tree.

Here is the general strategy for all instance method lookup:

- If method **myMethod()** exists in class **H**, then it is evaluated.
- Otherwise, JAVA checks the superclass of **H** for **myMethod** (in this case class **F**).
- If not found there, JAVA continues looking up the hierarchy until **Object** is reached, visiting additional classes **C**, **A** and **Object**.



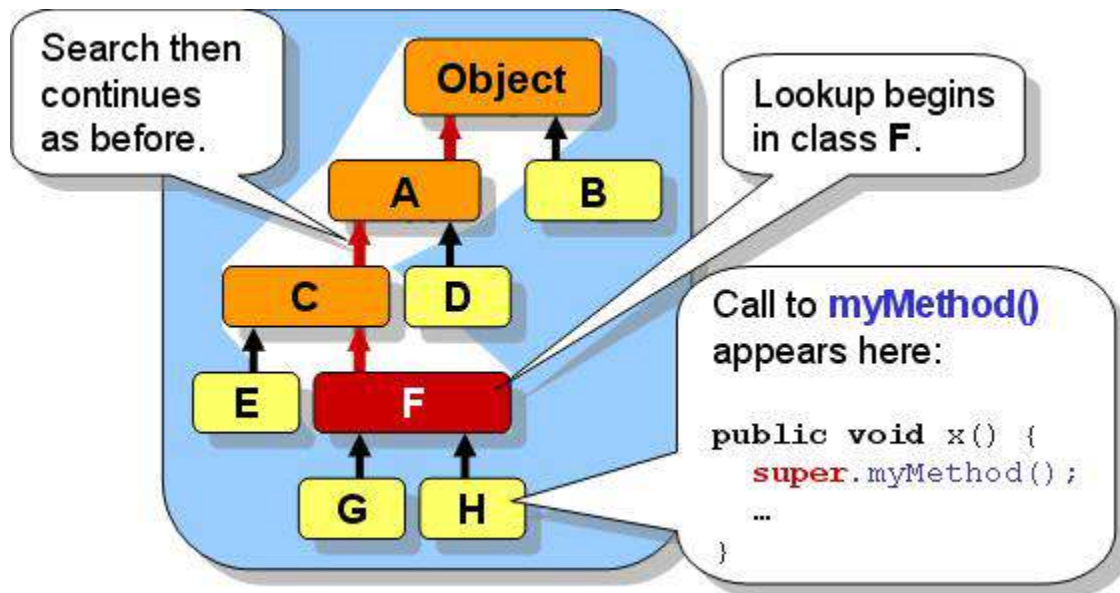
If not found at all during this search up to the **Object** class, the compiler will catch this and inform you that it cannot find method **myMethod()** for the object you are trying to sending it to:

```
C:\Test.java:20: cannot resolve symbol
symbol  : method myMethod ()
```

If there were many implementations of **myMethod()** along the path in the hierarchy (e.g., classes **F**, **C**, and **A** all implement **myMethod()**), then JAVA will execute the first one that it finds during its bottom-up search.

Notice the use of the keyword **this** in the picture. That tells JAVA to start looking for the method in "**this**" class. Alternatively, we can also use the keyword **super** here (i.e.,, **super.myMethod()**) to tell JAVA to *start* its search for the method in the *superclass*. If we used **super** in the example above, JAVA would start looking for **myMethod()** in class **F** first. If not found, it would then continue on up the tree looking for the method as usual.

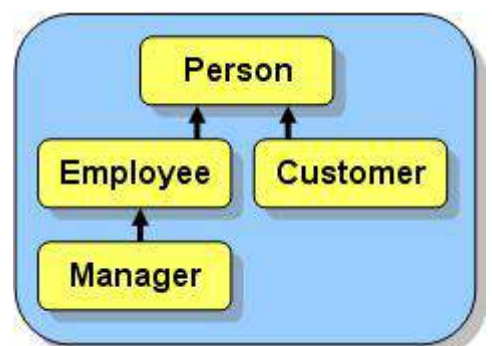
In fact if there was an implementation of `myMethod()` in the `H` class, it would not be called if we used `super`, since the search begins in the superclass, not in **this** class. So, the use of `super` merely specifies "where the method lookup should **begin** the search" ... nothing more.



How Are Access Modifiers Affected By Inheritance ?

It would be good to consider the effects that access modifiers have on attributes and methods within the class hierarchy. When an inherited attribute is declared as **private**, the subclasses still inherit it, but they cannot access it directly from within their own "local" code. For example, recall our previous example with **Customer**, **Manager** and **Employee** objects. Consider that all attributes are declared as **private**:

```
public class Person {
    private String name;
    private Address address;
    private String phoneNumber;
}
public class Employee extends Person {
    private int employeeNumber;
    private float hourlyPay;
}
public class Customer extends Person {
    private String[] itemsPurchased;
    private Date[] purchaseHistory;
}
public class Manager extends Employee {
    private String[] duties;
    private Employee[] subordinates;
}
```



Now consider the following code in this method written in the **Manager** class which determines whether or not a Manager has seniority. Assume that a manager has seniority if their employee number is less than 100 and they have more than 5 employees working for them.

```
public boolean hasSeniority() {
    return (employeeNumber < 100) && (subordinates.length > 5);
}
```

The code will NOT compile because the code is written in the **Manager** class but the inherited attribute **employeeNumber** is declared private within the **Employee** class. The **subordinates** attribute can be accessed without problems because it is defined in the same class as this method is written (i.e., the **Manager** class).

Since we need to access the **employeeNumber** attribute from the method ... how do we fix this? There are two solutions:

(1) write a public **getEmployeeNumber()** method in the **Employee** class and use it:

```
public class Employee extends Person {
    private int    employeeNumber;
    private float  hourlyPay;

    public int getEmployeeNumber() { return employeeNumber; }
}
```

```
public class Manager extends Employee {
    private String[]  duties;
    private Employee[] subordinates;

    public boolean hasSeniority() {
        return (getEmployeeNumber() < 100) && (subordinates.length > 5);
    }
}
```

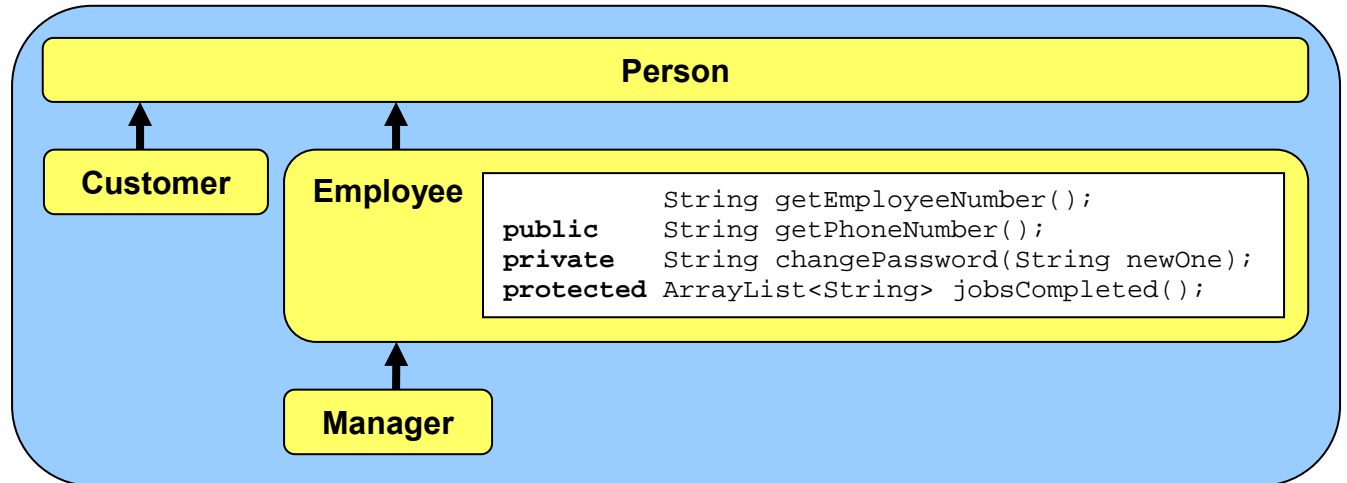
(2) declare all attributes that may need to be inherited as protected instead of private. By using protected, all subclasses can access the attribute directly, but no other classes may.

```
public class Employee extends Person {
    protected int    employeeNumber;
    protected float  hourlyPay;
}
```

```
public class Manager extends Employee {
    private String[]  duties;
    private Employee[] subordinates;

    public boolean hasSeniority() {
        return (employeeNumber < 100) && (subordinates.length > 5);
    }
}
```

Now how do **private** and **protected** modifiers affect methods ? Consider four methods within the **Employee** class with various access modifiers as follows:



Now consider some code within the **Manager** class that attempts to access these methods:

```

public class Manager extends Employee {
    public void tryThingsOut() {
        System.out.println(this.getEmployeeNumber()); // access allowed
        System.out.println(this.getPhoneNumber()); // access allowed
        System.out.println(this.changePassword("12345678")); // compile error
        System.out.println(this.jobsCompleted()); // access allowed
    }
}
  
```

Notice that the only method not allowed to be accessed is the **private** method, since the **tryThingsOut()** method is written in the **Manager** class, not in **Employee**.

Consider now the **Customer** class restrictions:

```

public class Customer extends Person {
    public void buyFrom(Employee emp) {
        System.out.println(emp.getEmployeeNumber()); // access allowed
        System.out.println(emp.getPhoneNumber()); // access allowed
        System.out.println(emp.changePassword("12345678")); // compile error
        System.out.println(emp.jobsCompleted()); // compile error
    }
}
  
```

Now we can no longer call the **jobsCompleted()** method, since it has been declared **protected** and **Customer** is not a subclass of **Employee**.

There is one more "protective" keyword that can be used with methods. We can declare a method as **final** to prevent subclasses from modifying the behavior. That is, when we declare a method as being final, JAVA prevents anyone from *overriding* that method. Hence no subclasses can have a method with that same name and signature:

```
public final void withdraw(float amount) {  
    ...  
}
```

Why would we want to do this ? Perhaps the behavior defined in the method is very critical and overriding this behavior "improperly" may cause problems with the rest of the program.

Restricting Class Access

In regards to class definitions, we are also allowed to indicate either *default* or **public** access to the class. So far, all of our classes have had **public** access, but we can have default access by leaving off the keyword **public**:

```
public class Manager { // public access from classes anywhere  
    ...  
}
```

```
class Employee { // default access from classes within package/folder  
    ...  
}
```

Interestingly, we can also declare a class as **final**. This means that it CANNOT have subclasses:

```
public final class Manager {  
    ...  
}
```

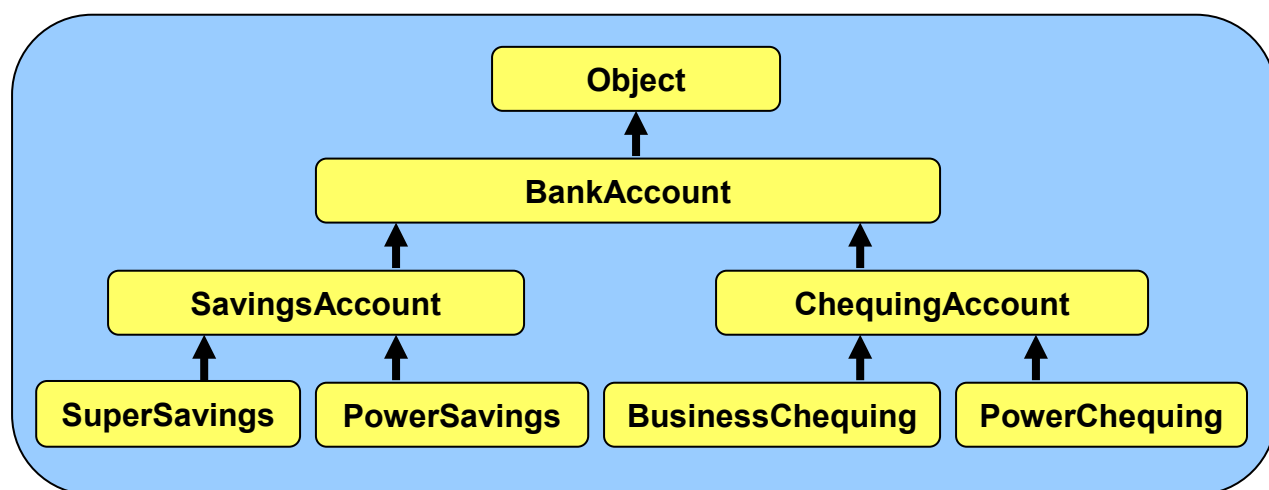
Why would we want to do this ? Perhaps the class has very weird code that the author does not want you to inherit ... maybe because it is too complicated and may easily be misused. Many of the JAVA classes (e.g., **ArrayList**) are declared as **final** which means that we cannot make any subclasses of them. It is a kind of security issue to prevent us from "messing up" the way those classes are meant to be used. It's a shame, because often we would like to have special types of **ArrayLists** and other similar objects.

4.3 Abstract Classes & Methods

Recall our example in the previous section pertaining to the various types of bank accounts. We had two types of accounts: **SuperSavings** and **PowerSavings**, which both inherited from a more general class called **SavingsAccount** and indirectly from **BankAccount** a little further up the hierarchy. Assume further that we distinguished between savings accounts and chequing accounts ... where chequing accounts allow their owners to write cheques.



Assume that the real bank actually has exactly 4 types of accounts so that when someone goes to the bank teller to open a new account, they specify whether or not they want to open a **SuperSavings**, **PowerSavings**, **BusinessChequing** or **PowerChequing** account. Here is a revised hierarchy ...



In our class hierarchy however, there are 7 account-related classes. The four classes representing the accounts that we can actually open are called **concrete** classes.

*A **concrete** class in JAVA is a class that we can make instances of directly by using the **new** keyword.*

That is, throughout our code, we will find ourselves creating one of these 4 classes. For example:

```

account1 = new SuperSavings(...);
account2 = new PowerSavings(...);
account3 = new BusinessChequing(...);
account4 = new PowerChequing(...);
  
```



However, we will likely never need to create instances of the other 3 account-related classes:

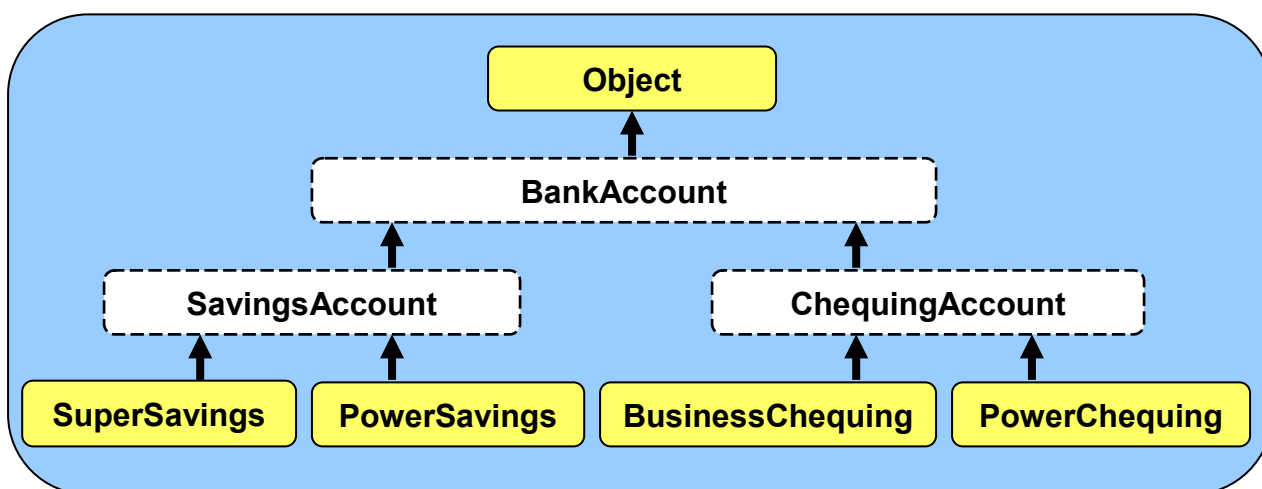
```

account5 = new BankAccount(...);
account6 = new SavingsAccount(...);
account7 = new ChequingAccount(...);
  
```



Why not? Well, put simply, these types of objects are not specific enough because they cause ambiguity. For example, if you went to the bank teller and asked to open just “a bank account”, the teller does not know which of the 4 types of accounts you actually want. The teller would likely ask you questions to help you narrow down your choices, but ultimately, the type of account that is opened (i.e., the account that is actually created) **MUST** be one of the 4 accounts that the bank offers. Likewise, in our program, if we were to create instances of **BankAccount**, **SavingsAccount** and **ChequingAccount**, then these objects would not be specific enough to define account behavior that matches one of the 4 real account types.

So in a sense, the **BankAccount**, **SavingsAccount** and **ChequingAccount** classes are not concrete, they are more abstract in that they don't exactly match the real-life objects. In JAVA, we actually use the term **abstract class** to define a class that we do not want to make instances of. So, **BankAccount**, **SavingsAccount** and **ChequingAccount** should all be **abstract** classes. We will draw abstract classes with dotted lines as follows ...



So, in JAVA ...

*An **abstract** class is a class for which we cannot create instances.*

That means, we can never call the constructor to make a new object of this type.

```

new BankAccount(...) // does not compile
new SavingsAccount(...) // does not compile
new ChequingAccount(...) // does not compile
  
```



All of the classes that we created so far in this course were concrete classes, although some could have been easily made abstract. We define a class to be abstract simply by using the **abstract** keyword in the class definition:

```

public abstract class BankAccount {
    ...
}
  
```

```
public abstract class SavingsAccount extends BankAccount {
    ...
}
```

```
public abstract class ChequingAccount extends BankAccount {
    ...
}
```

That is all that is involved in creating an abstract class. There really is nothing more to it. In fact, the remainder of the code in that class definition may remain as is.

So, in fact, by making a class abstract, all we have done is to prevent the user of the class from calling any of its constructors directly. This may raise an interesting question. If we cannot ever create new objects of the abstract class, then why would we ever want to create an **abstract** class in the first place ?

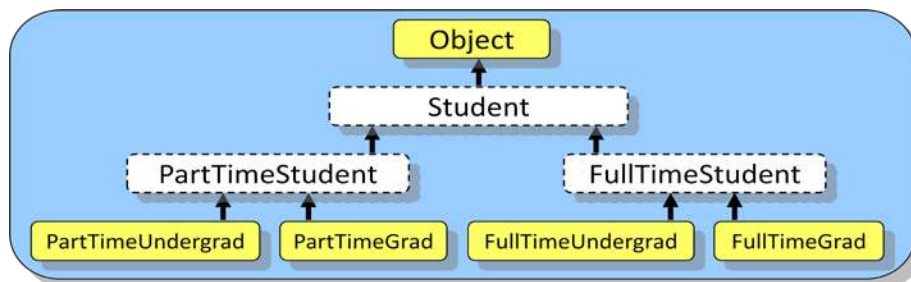
Well why did we create the **BankAccount** and **SavingsAccount** classes in the first place ? Inheritance was the key reason. These classes still contain the common attributes and shared behavior for all of their subclasses. The **BankAccount** class, for example, contains the 3 instance variables common to all accounts (**owner**, **accountNumber** and **balance**).

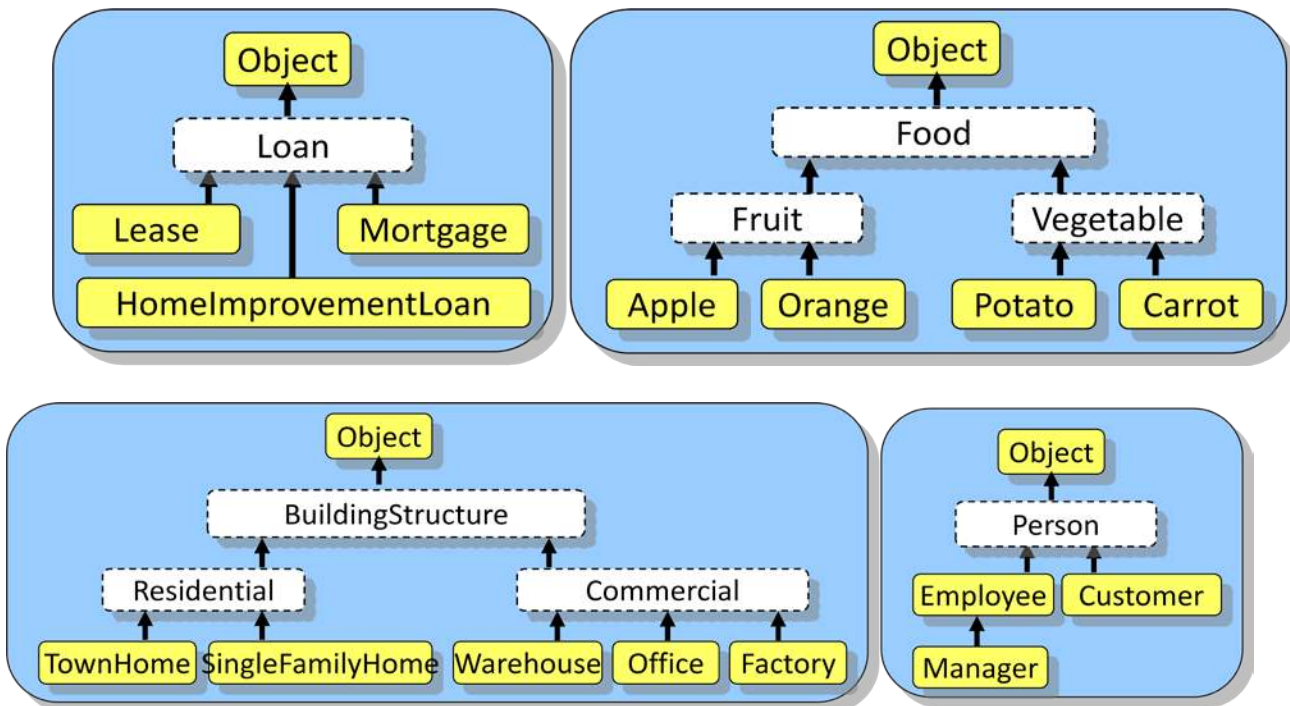
Also, the **SavingsAccount**, for example, contains the **deposit()** method that is shared between **SuperSavings** and **PowerSavings**. Hence, you can see that even though a class may be declared as **abstract** it is still useful and important in keeping our code organized properly in our class hierarchy. Their attributes and behaviors are still being used by their concrete subclasses.

How do we know which classes to make **abstract** and which ones to leave as **concrete** ? If we are not sure, it is better to leave them as concrete. However, if we discern that a particular class has subclasses that cover all of the possible concrete classes that we would ever need to create in our application, then it would be reasonable to make the superclass abstract.

Is there any advantage of making a class **abstract** rather than simply leaving it **concrete** ? Yes. By making a class **abstract**, you are informing the users of that class that they should not be creating instances of that class. In a way, you are telling them “**If you want to use this class, you should make your own concrete subclass of it.**”. You are actually *forcing* them to create a subclass if they want to use your abstract class. It forces the user of your class to be more specific in their object creation, thereby **reducing ambiguity** in their code.

Here are a few more examples of class hierarchies that we already discussed, showing how we could make some classes abstract:





Abstract Methods:

In addition to having **abstract** classes, JAVA allows us to make *abstract methods*:

An **abstract method** is a method with no code for which all concrete subclasses are **forced** to implement the method.

So, an abstract method is merely a specification of a method's signature (i.e., return type, name and list of parameters), but the body of the code remains blank. To define an abstract method, we use the **abstract** keyword at the beginning of the method's signature.

Here are a couple of examples:

```
public abstract void deposit(float amount);
public abstract void withdraw(float amount);
```

Notice that there are no braces **{ }** to specify the method body ... the method signature simply ends with a semi-colon **;**.

At this point you should be wondering: **“Why would any sane person would write a method that has no code in it ?”**. That is certainly a reasonable question since, after all, methods are called so that we can evaluate the code that is in them.

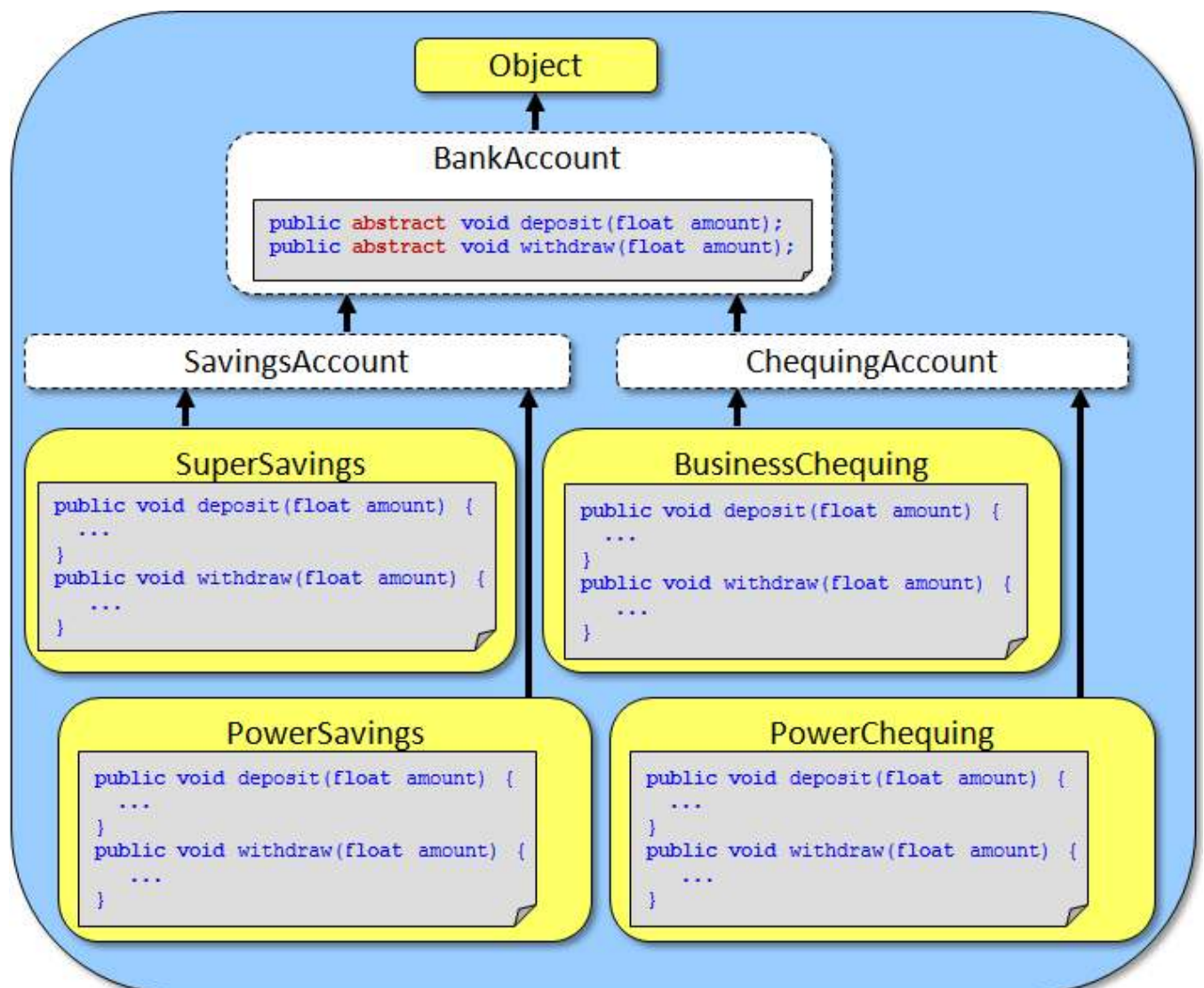
Abstract methods are actually never called, so JAVA never attempts to evaluate their code. Just as an abstract class is used to force the user of that class to have subclasses, an abstract method *forces* the subclasses to **implement** (i.e., to write



code for) that method. So, by defining an abstract method, you are really just informing everyone that the concrete subclasses must write code for that method. All concrete subclasses of an **abstract** class **MUST** implement the **abstract** methods defined in their superclasses, there is no way around it.

When JAVA compiles an abstract method for a class (e.g., class **A**), it checks to see whether or not all the subclasses of **A** have implemented the method (i.e., that they have written a method with the same return type, name and parameters). That is really all that happens in regard to the abstract methods.

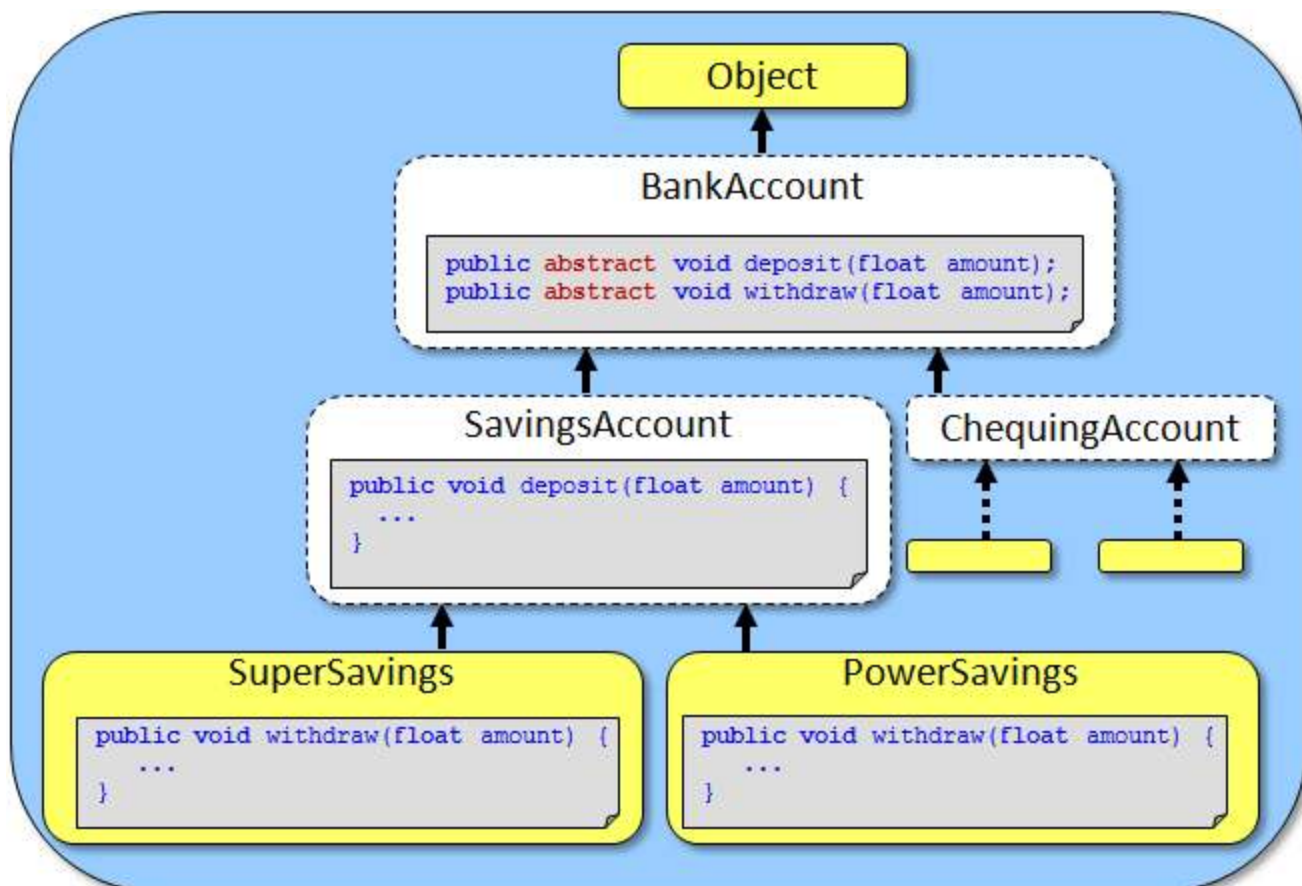
For example, if we make **deposit(float amount)** and **withdraw(float amount)** methods **abstract** in the **BankAccount** class, then, all of its concrete subclasses (**SuperSavings**, **PowerSavings**, **BusinessChequing** and **PowerChequing**) would be forced to implement those methods ... complete with code as follows ...



Each of the 4 concrete subclasses would implement their **deposit()** and **withdraw()** code according to the bank's rules for that type of account (i.e., apply certain fees, limit amount, etc...).

Alternatively, we can take advantage of inheritance. If, for example, the **SuperSavings** and **PowerSavings** accounts both **deposit()** in the same manner, instead of duplicating the code we can implement a non-abstract **deposit()** method in the **SavingsAccount** class that performs the required behavior. This method would then be shared (i.e., used) by both the **SuperSavings** and **PowerSavings** subclasses through inheritance.

In this case, the **SuperSavings** and **PowerSavings** classes would NOT need to implement the **deposit()** method, since it is inherited ...



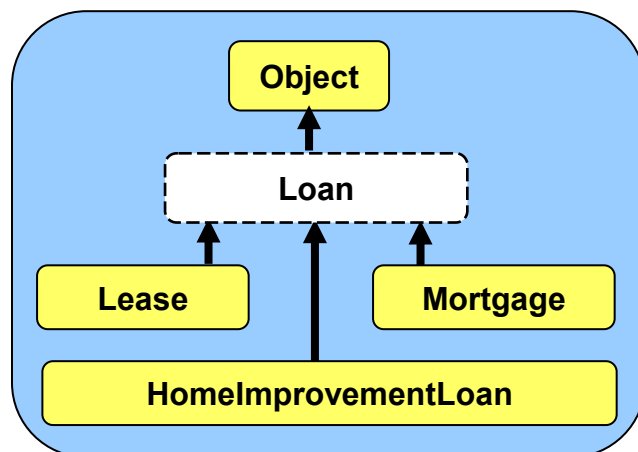
Only **abstract** classes are allowed to have such abstract methods. However, as you know, an **abstract** class may have regular methods as well.

If we were to find that all 4 types of concrete accounts did the exact same thing when a **deposit()** was made, then we would likely simply write the shared **deposit()** method in the **BankAccount** class, INSTEAD OF making the abstract **deposit()** method in the first place. This allows a kind of default **deposit()** behavior for all subclasses to inherit, not forcing any classes to implement this method.

It is often the case that we define more than one abstract method in a class. This allows us to **specify** a set of “standard” behavior that ALL of its subclasses MUST have.

For example, assume that we have the following hierarchy in which an abstract **Loan** class has 3 specific subclasses as shown here:

We may decide on some particular behavior that all types of loans must exhibit. For example, we may want to ensure that we have a way to calculate a monthly payment for the loan, a way to make payments on the loan, a way to re-finance the loan and perhaps a way to extract the client's information that pertains to the loan.



If this is the case, perhaps some of the behavior is similar for all loans (e.g., getting the client's information), while other behaviors may be unique depending on the type of loan (e.g., leases and mortgages may be re-financed differently). Here is how we might define the **Loan** class:

```

public abstract class Loan {
    public abstract float calculateMonthlyPayment();
    public abstract void makePayment(float amount);
    public abstract void renew(int numMonths);

    public Client getClientInfo() { // a non-abstract method
        ...
    }
    ....
}
  
```

Notice that the **getClientInfo()** method is non-abstract, so that we can write code in there that is shared by all the subclasses. The other 3 methods shown are **abstract** ... so the **Lease**, **Mortgage** and **HomeImprovementLoan** classes **MUST** implement all 3 of these methods, with the appropriate code. Remember ... an abstract class is just like any other class in regards to its attributes and behaviors. So there may be many more methods (abstract or non-abstract) and/or attributes defined in the **Loan** class.

Do you see the benefit of defining abstract methods? They allow you to define a set of behaviors that all your subclasses **must have** while giving them the flexibility to specify their **own unique code** for those behaviors. What would happen if we did not make any of the methods abstract?:

```

public abstract class Loan {
    public float calculateMonthlyPayment(){ return 0;}
    public void makePayment(float amount){ }
    public void renew(int numMonths){ }
    public Client getClientInfo() { ... }
    ....
}
  
```

Two things would be different. First, the methods would need to have a body. We could leave the code body blank or we could put in some default code of our choosing.

Second, the subclasses would not be “forced” to write these methods. So if the subclass did not supply the method, then these methods here would be inherited. This is not such a “big deal”, but if we simply *forgot* to implement these methods, then the inherited behavior may be unexpected and in some cases undesirable. By making the 3 methods **abstract**, the compiler will *force* us to write the methods, eliminating the possibility of us forgetting to implement them.

4.4 JAVA Interfaces

Inheritance allows all classes along the same path in the class hierarchy to share attributes and behaviors. The structure of the class hierarchy helps to identify common behavior that subclasses have with their superclasses. How though, would we define (and perhaps *force*) common behavior between seemingly *unrelated* classes in different parts of the class hierarchy?

There is a mechanism in JAVA for doing this:

*An **interface** is a specification (i.e., a list) of a set of methods such that any classes implementing the interface are forced to write these methods.*

Using an interface is similar to the idea of having a set of abstract methods, except that the interface exists on its own, that is, it is defined by itself in its own file.

We define such a list of methods as if we were defining a new class, except that we use the keyword **interface** instead of **class**:

```
public interface InterfaceName {  
    ...  
}
```

Just like classes, interfaces are *types* and are defined in their own .java files. So, the above interface would be saved into a file called **InterfaceName.java**.

Here is an example of an interface that defines a **Loanable** object:

```
public interface Loanable {  
    public float calculateMonthlyPayment();  
    public void makePayment(float amount);  
    public void renew(int numMonths);  
}
```

The methods themselves are defined like **abstract** methods, but without the word **abstract**. For comparison purposes, recall the similar **abstract** class called **Loan** with abstract methods:


```

public abstract class Loan {
    public abstract float calculateMonthlyPayment();
    public abstract void makePayment(float amount);
    public abstract void renew(int numMonths);

    public Customer getClientInfo() { // a non-abstract method
        //...
    }
    //....
}

```

There are some **similarities** between the two:

- both define three similar methods with no code.
- like abstract classes, we cannot create instances of interfaces. So, we cannot use this code anywhere in our code: `new Loan()` nor `new Loanable()`

There are also some **differences** between the two:

- We cannot declare/define any attributes nor **static** constants in an **interface**, whereas an **abstract** class may have them
- We can only declare “empty” methods in an **interface**, we cannot supply code for them. In contrast, an **abstract** class can have **non-abstract** methods with complete code.
- All methods in an **interface** must be declared **public**

Since interfaces are defined by themselves in their own files (i.e., the interface does not “belong” to any particular class), we must have a way to inform JAVA which objects will be implementing the methods that are defined in the interface.

Consider defining an interface called **Insurable** that defined the common behavior that all insurable objects MUST have as follows:

```

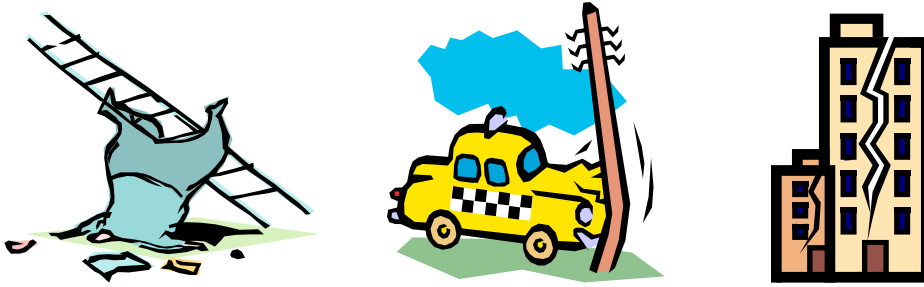
public interface Insurable {
    public int getPolicyNumber();
    public int getCoverageAmount();
    public double calculatePremium(int days);
    public java.util.Date getExpiryDate();
}

```

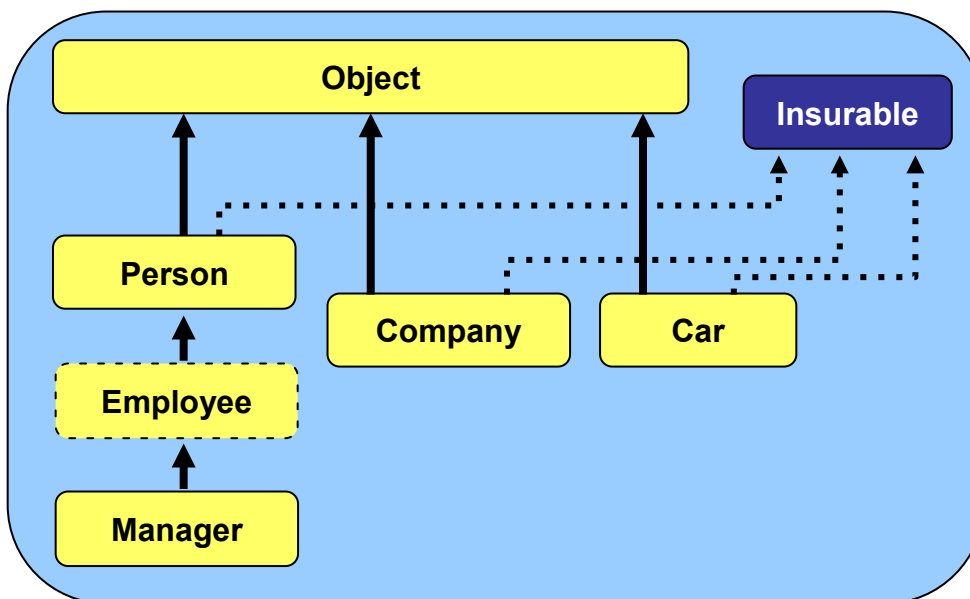


The code above would need to be saved and compiled before we can use it.

Assume now that we want to have some classes in our hierarchy that are considered to be insurable. Perhaps **Person**, **Car** and **Company** objects in our application are all considered to be *Insurable* objects.



We would want to make sure that they all implement the methods defined in the **Insurable** interface as shown here:



To do this in JAVA, we simply add the keyword **implements** in the class definition, followed by the **name** of the interface that the class will implement as follows:

```
public class Person implements Insurable {
    ...
}
```

```
public class Company implements Insurable {
    ...
}
```

```
public class Car implements Insurable {
    ...
}
```

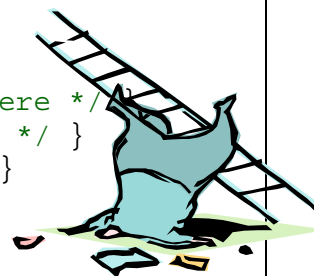
By adding this to the top of the class definition, we are informing the whole world that these objects are insurable objects. It represents a "stamp of approval" to everyone that these objects are able to be insured. It provides a "guarantee" that these classes will have all the

methods required for insurable items (i.e., `getPolicyNumber()`, `getCoverageAmount()`, `calculatePremium()` and `getExpiryDate()`). So then, for each of the implementing classes, we must go and write the code for those methods:

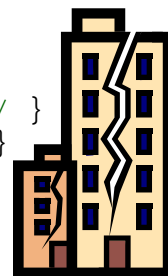
```
public class Car implements Insurable {
    //...
    public int    getPolicyNumber() { /* write code here */ }
    public double calculatePremium(int days) { /* write code here */ }
    public java.util.Date getExpiryDate() { /* write code here */ }
    public int    getCoverageAmount() { /* write code here */ }
    //...
}
```



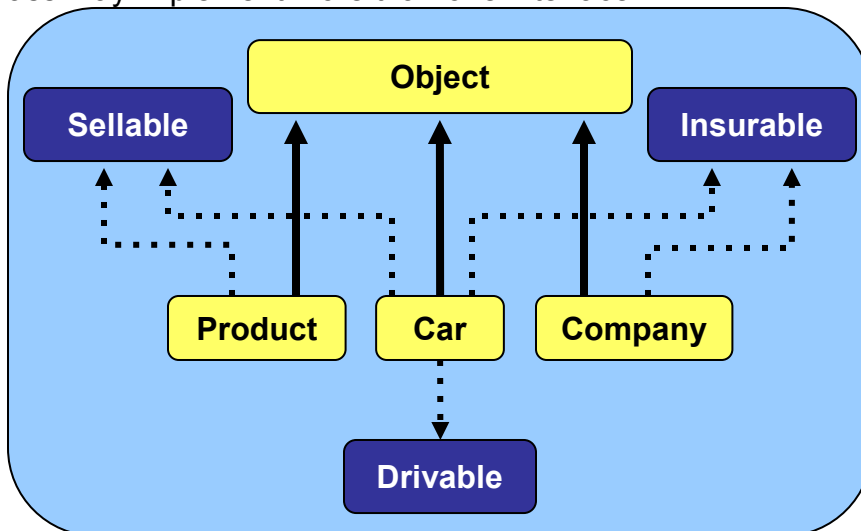
```
public class Person implements Insurable {
    //...
    public int    getPolicyNumber() { /* write code here */ }
    public double calculatePremium(int days) { /* write code here */ }
    public java.util.Date getExpiryDate() { /* write code here */ }
    public int    getCoverageAmount() { /* write code here */ }
    //...
}
```



```
public class Company implements Insurable {
    //...
    public int    getPolicyNumber() { /* write code here */ }
    public double calculatePremium(int days) { /* write code here */ }
    public java.util.Date getExpiryDate() { /* write code here */ }
    public int    getCoverageAmount() { /* write code here */ }
    //...
}
```



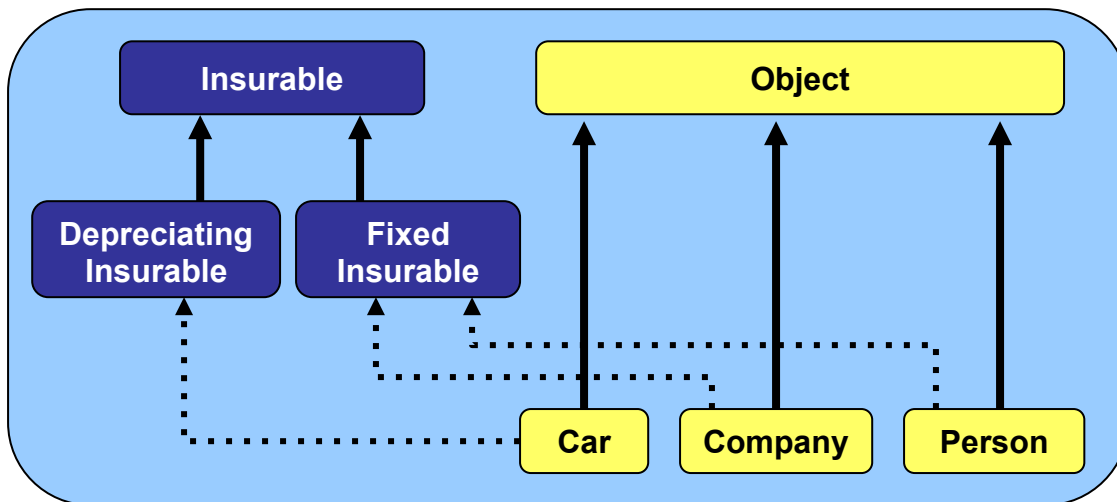
Remember that these classes may define their own attributes and methods but somewhere in their class definition they must have ALL 4 methods listed in the **Insurable** interface. Interestingly, a class may implement more than one interface:



Here the **Car** object implements 3 interfaces. To allow this in our code, we just need to specify each implemented interface in our class definition (in any order), separated by commas:

```
public class Car implements Insurable, Drivable, Sellable {
    ...
}
```

Of course, the **Car** class would have to implement **ALL** of the methods defined in **each** of the three interfaces. Like classes, interfaces can also be organized in a hierarchy:



As with classes, we form the interface hierarchy by using the **extends** keyword:

```
public interface Insurable { ...
    public int getPolicyNumber();
    public int getCoverageAmount();
    public double calculatePremium(int days);
    public java.util.Date getExpiryDate();
}
```

```
public interface DepreciatingInsurable extends Insurable {
    public double computeFairMarketValue();
    public void amortizePayments();
}
```

```
public interface FixedInsurable extends Insurable {
    public int getEvaluationPeriod();
}
```

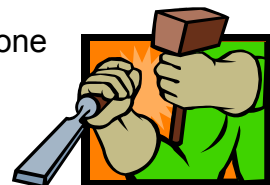
Classes that implement an interface must implement its "super" interfaces as well. So **Company** and **Person** would need to implement the method in **FixedInsurable** as well as the four in **Insurable**, while **Car** would have to implement the two methods in **DepreciatingInsurable** and the four in **Insurable** as well.

In summary, how do interfaces help us ? They provide us with a way in which we can specify common behavior between arbitrary objects so that we can ensure that those objects have specific methods defined. There are many pre-defined interfaces in JAVA and you will see them used often when we discuss user interfaces.

4.5 Polymorphism

Recall that we can convert (or type-cast) primitives to convert a value from one type to another:

```
(int) 871.34354;    // results in 871
(char) 65;         // results in 'A'
(long) 453;       // results in 453L
```



Some type-casting is done automatically by JAVA when we assign a value of one particular type to a variable of a different type. However, we can also explicitly type-cast in order to simplify the data (e.g., from **float** to **int**) or for display purposes (e.g., from **byte** to **char**).

In JAVA, we can also type-cast **objects** from one type to another type. However, type-casting objects is **different** from type-casting primitives in that the objects are **not converted or modified** in any way. Instead, when we type-cast an object variable, it is simply restricted with respect to the kinds of behaviors that it is capable of doing from then on in our program.

Why would we want to do type-casting if all that we are doing is restricting the object in some way. Would it not be better (i.e., more flexible) to simply allow the object's methods to be used at any time ? These are valid questions. However, there are reasons for type-casting.

Perhaps the main advantage of type-casting is that it allows for:

Polymorphism is the ability to use the same behavior for objects of different types.

That is, it allows different objects to respond to the exact "same" methods. The result is that we have much less to remember when we go to use the object. That is, by using polymorphism, we just need to understand a few commonly used methods that all these objects understand. For example:



- We can ask all **Person** objects what their **name** is. This is independent as to whether or not they are instances of **Employees**, **Managers**, **Customers** etc...
- We can deposit to any **BankAccount**, independent of its type.

And so ... by treating an object more generally (i.e., type-casting it), we are simplifying the way that we will use the object by restricting its usage to a few well understood methods. As a result, our code becomes

- easier to understand
- more intuitive and
- quicker to write since the programmer does not need to remember as many methods.

It is important to understand the type-casting of objects because JAVA often type-casts objects automatically. Therefore, we must understand *how* to type-cast and *when* it is done automatically. The type-casting of objects is done the same way (i.e., with the round brackets) as with primitives. Here are a few examples:

```
p = (Person)anEmployee;
c = (Customer)anArray[i];
b = (SavingsAccount)aBankAccount;
```

Notice that there is an object type (i.e., class name) within the round brackets.

When we type-cast an object to another type we are not modifying it in any way. Rather, we are simply causing the object to be “**treated**” more generally from then on in the program. As a result, the object will then be less flexible in that we can no longer call some of the methods that we used to call on it. In a way, we are ignoring some of the behavior that is available to the object.



This may sound strange, but we do this in real life. Let us consider a couple of examples.

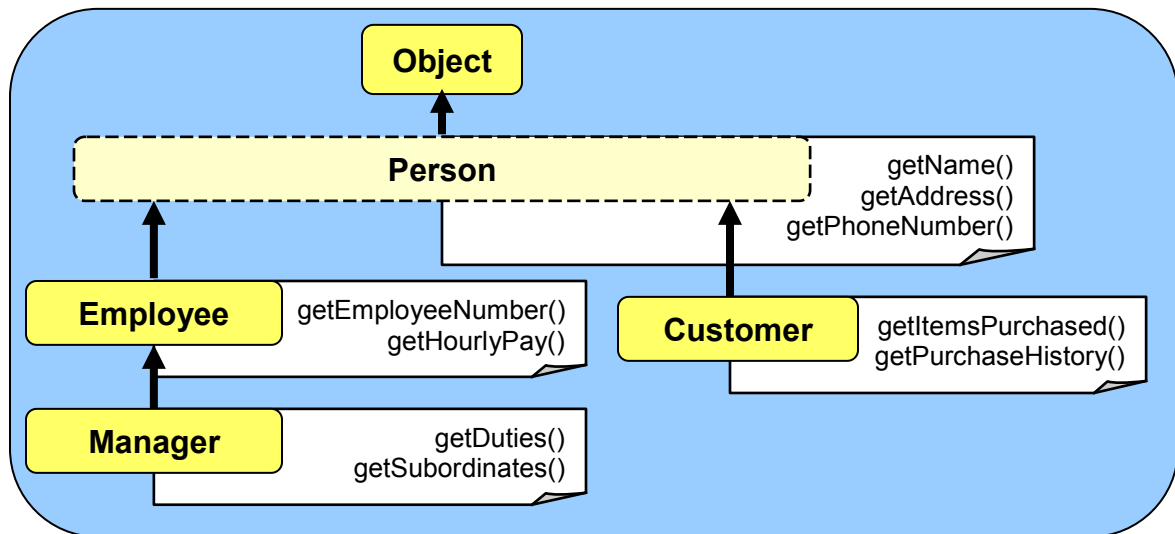


Consider meeting your professor with his family outside of class, perhaps at a local shopping mall. Likely, you would “treat” your professor as a general/normal **Person** ... not as your “professor”. So, you might ask him questions that you would ask anyone such as: “*Is this your family?*” or “*What are you shopping for today?*”. However, you would likely not ask him a question like “What kind of questions will be on the final exam?” and hopefully you would not pull out a laptop and ask him to help you debug the code on your assignment. So, in a sense, you have type-casted the **Professor** to a more general **Person** object by restricting the available behaviors to those that are applicable to more general people, avoiding any professor-specific behavior.



As another example, consider an **Apple** ... normally you may **polish**, **peel** or **eat** it ... but in a food fight, you may type-cast (i.e., treat) your apple as a general **throwable** projectile. Then, the apple takes on different behavior such as **throw**, **catch**, **splatter**, etc... The fact is ... it is still an **Apple**, but it is being *treated* differently. You may even type-cast other objects to be projectiles such as grapes, sandwiches, pineapples (ouch), chairs, etc...

Now let us look at a real coding example. Consider the following class hierarchy of **Employee**, **Person**, **Manager** and **Customer** objects with some instance methods belonging to each class as shown:



Consider what happens when we create a single **Employee** object and then type-cast it to a **Person**. Take note of the methods that are available for use and those which will not compile. Note that we create 2 variables, yet both point to the same object ...

```

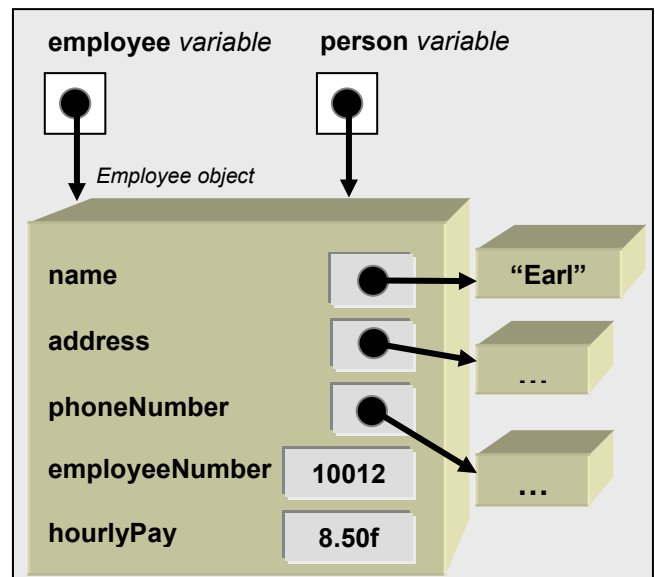
Person    person;
Employee  employee;

employee = new Employee("Earl");
employee.getName();
employee.getAddress();
employee.getPhoneNumber();
employee.getEmployeeNumber();
employee.getHourlyPay();

// now treat Earl like a person
person = (Person)employee;
person.getName();
person.getAddress();
person.getPhoneNumber();

// these two will not compile
person.getEmployeeNumber();
person.getHourlyPay();

// type-cast back and all is ok
((Employee)person).getEmployeeNumber();
((Employee)person).getHourlyPay();
  
```



You will notice that once the type-cast to **(Person)** occurs, we are no longer able to use the **getEmployeeNumber()** and **getHourlyPay()** methods since they are **Employee**-specific

methods and we are now treating Earl as simply a **Person**. However, the **person** variable is still pointing to Earl ... the exact same object.

When we type-cast the **person** variable back to (**Employee**) again, and then try the same two methods, they work fine because we are now treating Earl as an **Employee** again.

Notice what we are **not** able to do:

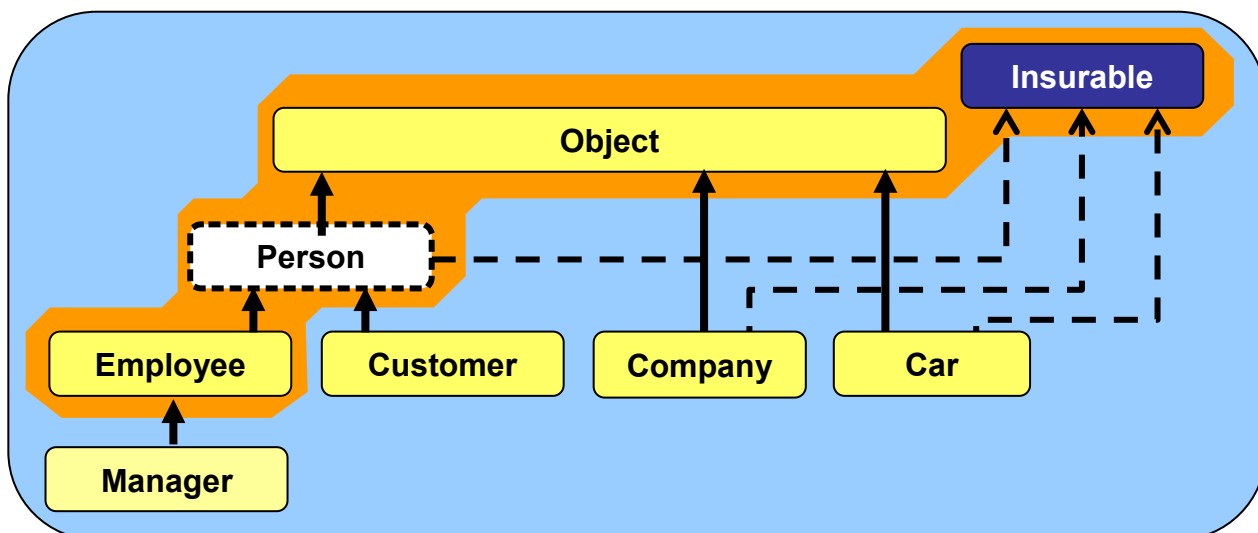
```
Employee   employee;
Manager    manager;
Customer   customer;

employee = new Employee("Earl");
manager = (Manager)employee;    // Type-cast is not allowed
customer = (Customer)employee;  // Type-cast is not allowed
```

We are only allowed to use class type-casting to generalize an object. Therefore we can only type-cast to classes up the hierarchy (e.g., **Person** and **Object**) but not down the hierarchy (e.g., **Manager**) or across the hierarchy (e.g., **Customer**) from the original object class (e.g., **Employee**). In summary, objects may **ONLY** be type-casted to:

- a type which is one of its **superclasses**
- an **interface** which the class implements
- or back to their own class again

In the following example, an **Employee** object can *only* be type-casted to (or stored in a variable of type) **Employee**, **Person**, **Object** or **Insurable**:



Attempts to type-cast to anything else will generate a **ClassCastException**. So **Employees** CANNOT be type-casted to **Manager**, **Customer**, **Company** or **Car**. Such restrictions make sense, after all, why would we "treat" a **Manager** as a **Company** or a **Car**.

Some coding advantages arise through *implicit* or *automatic* type-casting. Sometimes JAVA will automatically type-cast an object, even if we do not explicitly do so with the brackets ().

There are two main situations in which automatic type-casting occurs:

1. when we assign an object to a variable with a more general type:

```
Person    person;
Employee  employee;

employee = new Employee("Earl");
person = employee;    // same as person = (Person)employee;
```

2. when we pass in the object as a parameter to a method which has a more general type:

```
Employee  employee;

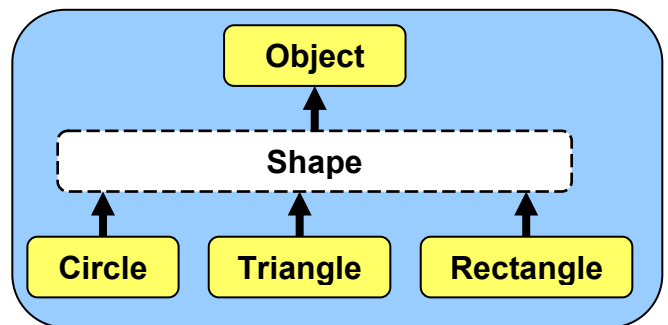
employee = new Employee("Earl");
doStandardHiringProcess(employee);
...
```

```
public void doStandardHiringProcess(Person p) {
    // employee object is type-casted to Person upon entering method
    ...
}
```

In both cases, you should be aware that an automatic type-cast has taken place. In fact, it usually does not matter if you “know” that the type-casting is taking place, because the compiler will tell you. However, it tells you this by means of a compile error ... which is somewhat unpleasant, as you well know. Also, sometimes the compiler message is not straightforward to understand.

Let us now look at a simple example to see how much we can reduce our code through the use of automatic type-casting. Consider a hierarchy of shape-related objects as shown here. We can create a **Circle**, a **Triangle** and a **Rectangle** and all three can be stored into a variable of type **Shape**:

```
Shape    s;
Circle   c = new Circle(20);
Triangle t = new Triangle(10, 20, 30);
Rectangle r = new Rectangle(10, 10, 20, 20);
s = c;    // s points to object c
s = t;    // s points to object t
s = r;    // s points to object r
```



Notice that we did not make any explicit type-cast to **Shape** (although we could have done so). Here we simply re-assigned variable **s** to have three different values corresponding to three different types of objects. The example code itself is pointless, but it helps us to see how we can use automatic type-casting.



Assume now that we want to draw a shape and that the **Circle**, **Triangle** and **Rectangle** classes all have an appropriate method for drawing themselves called **draw()**:

```
public class Circle extends Shape {
    ...
    public void draw() { ... }
}
```

```
public class Triangle extends Shape {
    ...
    public void draw() { ... }
}
```

```
public class Rectangle extends Shape {
    ...
    public void draw() { ... }
}
```

Consider now our **Shape** variable **s** which can hold any kind of shape:

```
Shape s = ...;
```

At any given time, we may not know exactly which kind of shape is currently stored in the **aShape** variable. How then do we know which **draw()** method to call? Well, we could check the type of the object, perhaps with the **instanceof** keyword and then use some **if** statements as follows:

```
if (s instanceof Circle)
    s.draw();

if (s instanceof Triangle)
    s.draw();

if (s instanceof Rectangle)
    s.draw();
```



However, looking at the code, it is clear that regardless of the type of shape we have, we just need to call **draw()**. Since we called all of the methods **draw()**, this is an example of polymorphism ... that is ... all shape objects understand the **draw()** method. For this to compile though, there should also be a **draw()** method defined in the **Shape** class, which may be blank.

As a result, because of polymorphism and the explicit type-cast, we don't even need the **IF** statements. Our code can be simplified to:

```
s.draw();
```

Incredible!!! What a reduction in code! But why does this work? How does JAVA know which **draw()** method to call? Well, remember, whatever we store in the **Shape** variable **s** does not change its type. The compiler will look at the kind of object that we put in there and call the appropriate method accordingly by starting its method lookup in the class corresponding to that object type (i.e., either **Circle**, **Triangle** or **Rectangle**, depending on what was stored in **s**). As you can see, polymorphism can be quite powerful.

Now consider a **Pen** object which is capable of drawing shapes. We would like to use code that looks something like this:

```
Pen aPen = new Pen();  
  
aPen.draw(aCircle);  
aPen.draw(aTriangle);  
aPen.draw(aRectangle);
```



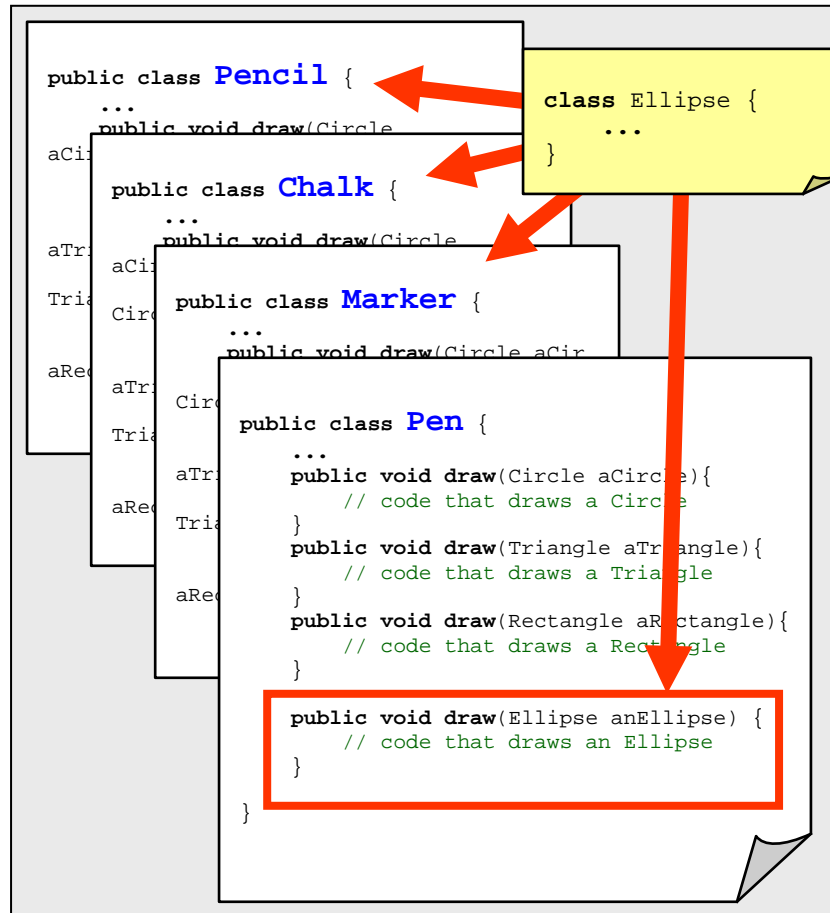
However, this is not so straight forward. We would have to define a **draw()** method in the **Pen** class for each kind of shape in order to satisfy the compiler with regards to the particular type of the parameter:

```
public class Pen {  
    ...  
    public void draw(Circle aCircle) {  
        // code that draws a Circle  
    }  
    public void draw(Triangle aTriangle) {  
        // code that draws a Triangle  
    }  
    public void draw(Rectangle aRectangle) {  
        // code that draws a Rectangle  
    }  
}
```

Since the drawing code is likely different for all 3 shapes we will need the 3 different pieces of code to do the drawing. However, all of the shape-drawing code must appear here in the **Pen** class. This is somewhat intuitive in regards to real life, since **Pen's** draw shapes.

However, if we had other drawing classes such as **Pencil**, **Marker** or **Chalk**, we would need to go to all these classes and insert shape-specific code for each kind of shape. Even worse, if we wanted to add shapes (e.g., **Ellipse**, **Diamond**, **Parallelogram**, **Rhombus**, etc..) then we would have to go to the **Pen**, **Pencil**, **Marker** and **Chalk** classes to add the appropriate shape-drawing code.

This is quite terrible since our code is not modular ... the adding of one simple **Shape** class would require us to recompile 4 other classes.



There must be a better way to do this! The answer is to use a technique known as **double-dispatching**. When we call a method in JAVA, this is the same notion as *sending a message* to the object. The idea behind double-dispatching is to *dispatch* a JAVA message two times. Through double dispatching, we force a second message to be sent (i.e., we call another method) in order to accomplish the task.

Before we do the double-dispatch, we need to adjust our code a little. We can simplify the **draw()** methods in the **Pen**, **Pencil**, **Marker** and **Chalk** classes by combining them all in one method. The new method will take a single parameter of type **Shape**. Hence, through type-casting, we can pass in any subclass of **Shape** to the method. Here is the code ...

```

public class Pen {
    ...
    public void draw(Shape anyShape) {
        if (anyShape instanceof Circle)
            // Do the drawing for circles
        if (anyShape instanceof Triangle)
            // Do the drawing for triangles
        if (anyShape instanceof Rectangle)
            // Do the drawing for rectangles
    }
}

```

At this point, we still have to decide how to draw the different **Shapes**. So then when new **Shapes** are added, we still need to come into the **Pen** class and make changes. However, we can correct this problem by shifting the drawing responsibility to the individual shapes themselves, as opposed to it being the **Pen's** responsibility. This "shifting" (or flipping) of responsibility is where the notion of **double dispatching** comes in. It is similar to the expression "passing-the-buck" in English. In other words, we are saying: **"I'm not going to do it ... you do it yourself"**.

We perform double-dispatching by making a method in each of the specific **Shape** subclasses that allows the shape to draw itself using a given **Pen** object:

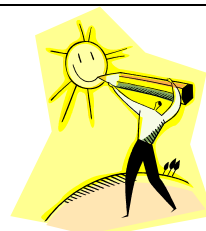
```
public class Circle extends Shape {
    ...
    public void drawWith(Pen aPen) { ... }
}
```

```
public class Triangle extends Shape {
    ...
    public void drawWith(Pen aPen) { ... }
}
```

```
public class Rectangle extends Shape {
    ...
    public void drawWith(Pen aPen) { ... }
}
```

Then, we do the double dispatch itself by calling the **drawWith()** method from the **Pen** class:

```
public class Pen {
    ...
    public void draw(Shape aShape) {
        aShape.drawWith(this);
    }
}
```

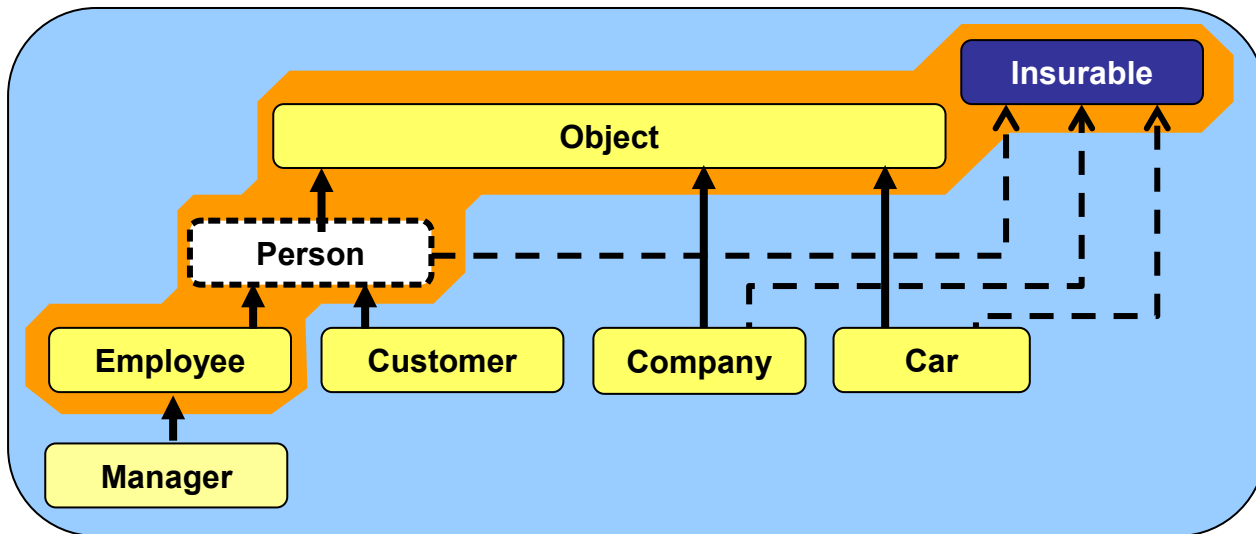


Notice that the code is incredibly simple. When the **Pen** is asked to draw a **Shape**, it basically says: "No way! Let the shape draw itself using ME!". That is the second message call, which itself does the real drawing work. We would write a similar one-line method in the **Pencil**, **Chalk** and **Marker** classes. In order for this to compile, you must also have a **drawWith(Pen aPen)** method declared in class **Shape** even if that method does nothing.

Do you see the tremendous advantages here? Regardless of the kind of **Shape** that we may add in the future, we NEVER have to go into the **Pen**, **Pencil**, **Marker** or **Chalk** classes to make changes. This code remains intact. Instead, we simply write a **drawWith()** method in the new **Shape** class to do the drawing of itself. And who would know better how to draw the

shape than itself. The code is much more modular and has a nice clean separation. Furthermore, the code is logical and easy to understand.

Type-casting also provides advantages when multiple unrelated classes implement the same interface. Objects can be type-casted to an interface type, provided that the class implements that interface. In the hierarchy below, we can type-cast any instances of **Car**, **Company**, **Customer**, **Employee** or **Manager** to **Insurable**.



Assume that **Insurable** has a method defined called **getPolicyNumber()** and that the **Car** class has a **getMileage()** method. Notice the type-casting as follows:

```

Car      jetta = new Car();
Insurable item = (Insurable)jetta;

item.getPolicyNumber(); // OK since Insurable
jetta.getMileage();     // OK (assuming it is a Car method)
item.getMileage();     // Compile Error
((Car)item).getMileage(); // OK now
  
```

Notice the compile error when calling **getMileage()** on **item**. Even though **item** is actually a **Car** object, it has been type-casted to **Insurable**, and so only methods that are defined in the **Insurable** interface can be used on it.

What is the advantage of type-casting to an interface? Well, we can treat “seemingly unrelated” objects the same way. This is often useful when we have a collection of such items.

Consider an Array of a variety of **Insurable** items and then trying to list all of the policies and totaling the amounts of all the policies:

```

float          total = 0;
Insurable[]   insurableItems;

insurableItems = new Insurable[5];
insurableItems[0] = new Car("Porshce", "Carerra", "Red", 340);
insurableItems[1] = new Customer("Guy Rich");
insurableItems[2] = new Company("Elmo's Edibles", 2009);
insurableItems[3] = new Employee("Jim Socks");
insurableItems[4] = new Manager("Tim Burr");

System.out.println("Here are the policies:");
for (int i=0; i<insurableItems.length; i++) {
    System.out.println("  " + insurableItems[i].getPolicyNumber());
    total += insurableItems[i].getPolicyAmount();
}
System.out.println("Total policies amount is $" + total);

```

In the above example, all 5 unique objects are automatically type-casted to **Insurable** when added to the array. Then when listing the policies, we simply use the common **getPolicyNumber()** method (which must be defined in **Insurable** and implemented by all the classes). Similarly, we total all the policy amounts by using the common **getPolicyAmount()** method.

What would the code look like without having the **Insurable** interface? Well, in order to store the items in the same array we would still need to know what they have in common. Without the **Insurable** interface, the only other thing that all the objects have in common is that they are subclasses of **Object**. So we would have to make an **Object[5]** array of general objects: `Object[] insurableItems = new Object[5];`

Once we make these changes, then the compiler will prevent us from calling the **getPolicyNumber()** or **getPolicyAmount()** methods because it assumes that the **item** extracted in the FOR loop is a general **Object** ... but general objects do not have such methods. Therefore, we would be forced to check the type of every object, beforehand ... implying that we knew all the different types that would ever be placed in the array. Our code would be longer, more complicated, messier and non-modular:

```

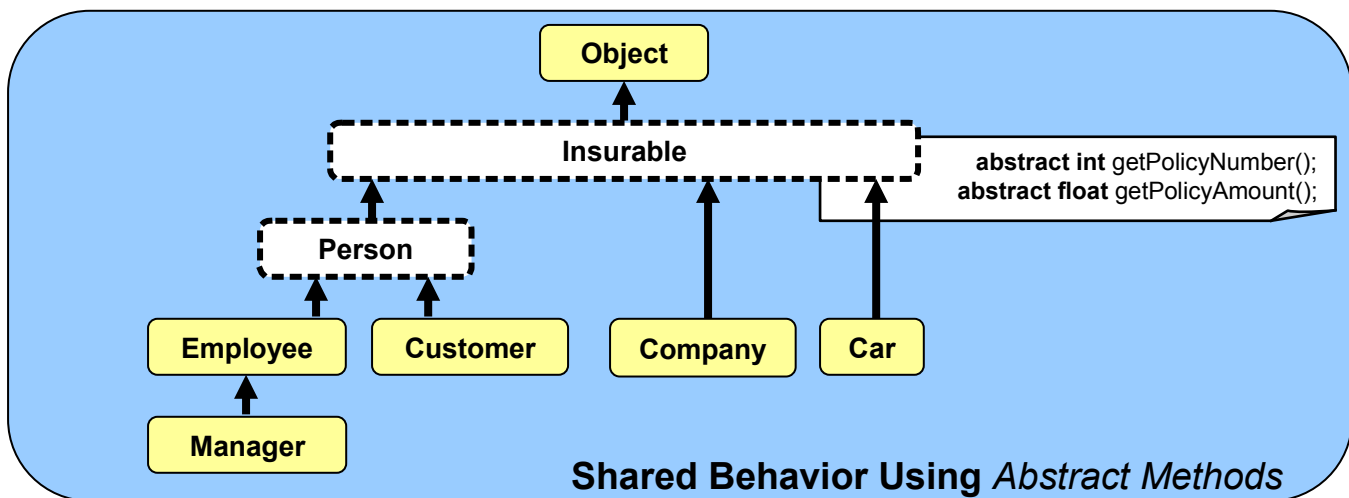
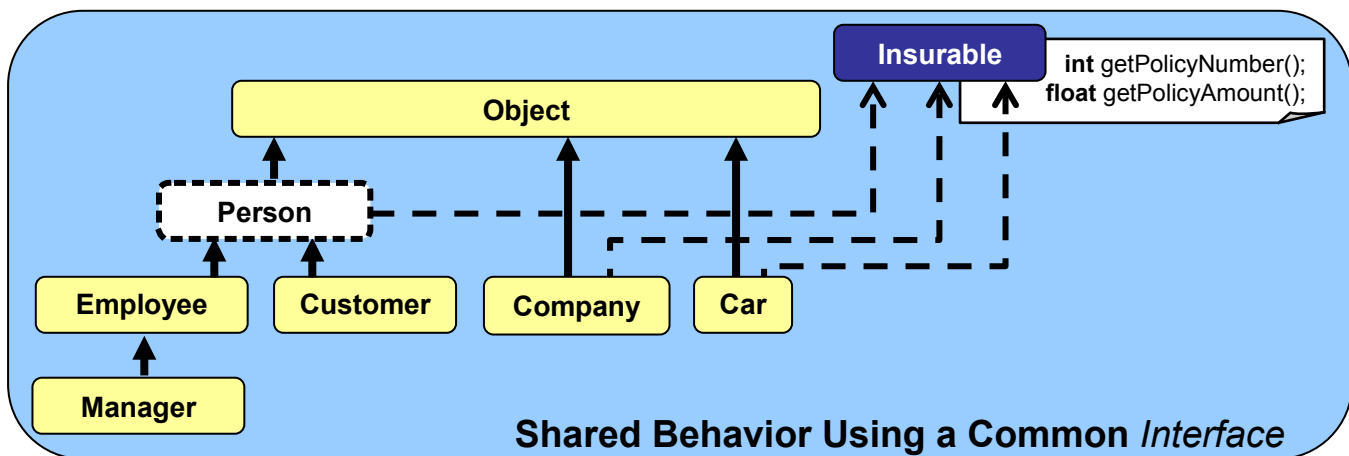
...
for (int i=0; i<insurableItems.length; i++) {
    if (insurableItems[i] instanceof Car)
        System.out.println("  " + ((Car)insurableItems[i]).getPolicyNumber());
        total += ((Car)insurableItems[i]).getPolicyAmount();
    }
    else if (insurableItems[i] instanceof Employee)
        System.out.println("  " + ((Employee)insurableItems[i]).getPolicyNumber());
        total += ((Employee) insurableItems[i]).getPolicyAmount();
    }
    else if (...)
        ...
}

```

Of course, an alternative to using the shared interface would be to have all insurable objects extend (i.e., inherit from) a common abstract class, perhaps called **Insurable** as well. We could then define the **getPolicyNumber()** and **getPolicyAmount()** methods as **abstract** methods, forcing all subclasses to implement them. Then, we could use the same identical code that worked with the **Insurable** interface.

However, the big disadvantage of doing things this way, is that we are restricting the inheritance of **Insurable** objects to be insurable-related. That means, we cannot take advantage of other kinds of inherited attributes and behaviors.

Here is a diagram showing how we could get such shared behavior either with interfaces or with abstract methods ...



As another more tangible example, consider defining a **Controllable** interface for objects that can be controlled via remote control. The interface may look as follows:


```
public interface Controllable {
    public void turnLeft();
    public void turnRight();
    public void moveForward();
    public void moveBackward();
}
```

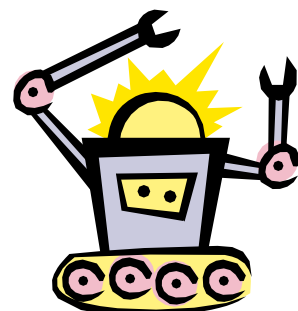


Now, consider a **Robot** object which is **Controllable** and implements this interface:

```
public class Robot implements Controllable {
    private int batteryLevel;
    private Behavior[] behaviors;

    // These are the Controllable-related methods
    public void turnLeft() { ... }
    public void turnRight() { ... }
    public void moveForward() { ... }
    public void moveBackward() { ... }

    // There will likely also be some other methods
    // which are robot-specific
    public Behavior computeDesiredBehavior() { ... }
    public int readSensor(Sensor x) { ... }
    ...
}
```



Now, what about a **ToyCar**, or even a **Lawnmower**? We can implement the **Controllable** interface for each of these as well. In fact, suppose that we want to set up a handheld remote control for **Controllable** objects. We can then treat all of the objects (**Robots**, **ToyCars**, **Lawnmowers**, etc...) as a single type of object ... a **Controllable** object:



```
public class RemoteControl {
    private Controllable machine;

    public RemoteControl(Controllable m) {
        machine = m;
    }

    public void handleButtonPress(int buttonNumber) {
        if (buttonNumber == 1)
            m.moveForward();
        else if (buttonNumber == 2)
            m.moveBackward();
        else if (buttonNumber == 3)
            m.turnLeft();
        else
            m.turnRight();
    }
    ...
}
```



Notice that the remote control constructor is supplied with any object that is of type **Controllable** (i.e., a **Robot**, **ToyCar**, **Lawnmower**, etc..) Therefore, as can be seen in the **handleButtonPress()** method, the code for controlling the machine from the remote is independent of the type of object being controlled.

This is a nice clean separation of code in that any new **Controllable** object that is developed in the future can be controlled by this **RemoteControl** object. The programmer would not need to make any changes to the **RemoteControl** class code whatsoever:

```
ToyPlane    aPlane = new ToyPlane();
ToyBoat     aBoat  = new ToyBoat();

RemoteControl planeRemote = new RemoteControl(aPlane);
RemoteControl boatRemote  = new RemoteControl(aBoat);
```

Chapter 5

Graphical User Interfaces

What is in This Chapter ?

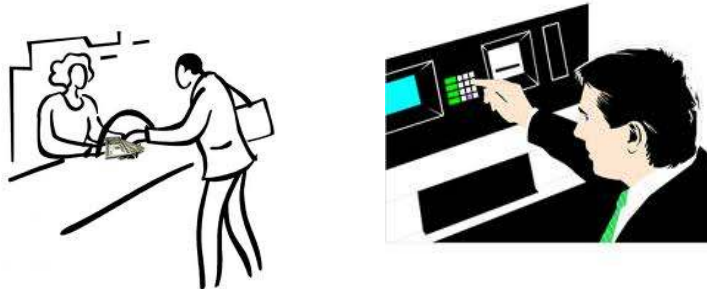
This chapter discusses developing JAVA applications that bring up windows that the user can interact with. It discusses **Graphical User Interfaces** and how we can develop our own to represent main windows for our JAVA applications.



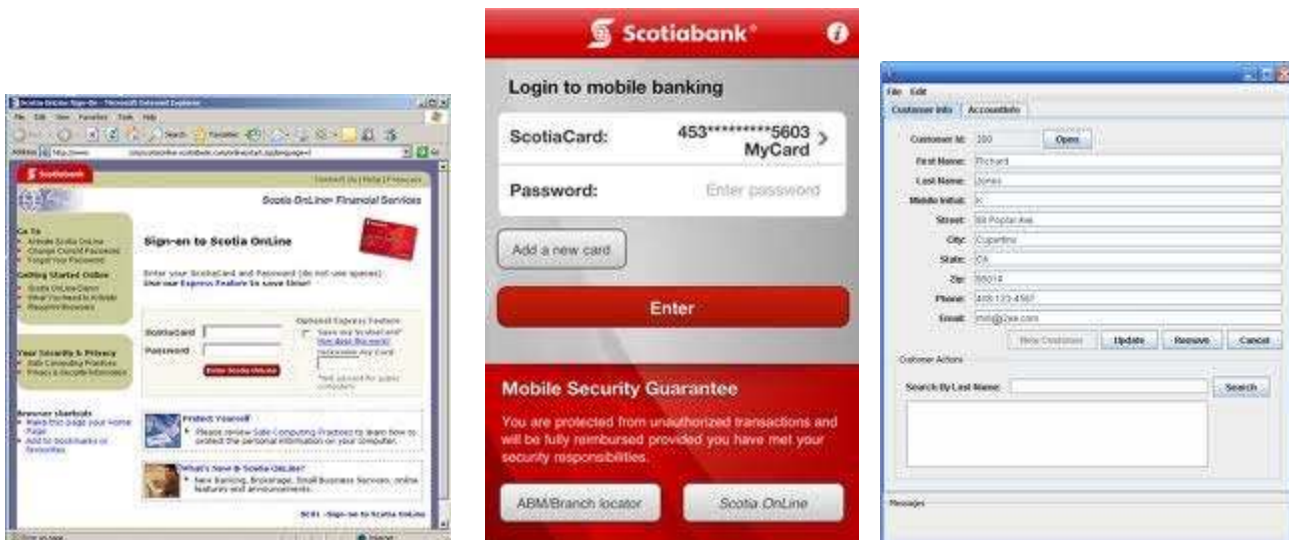
5.1 User Interfaces

All applications require some kind of **user interface** which allows the user to interact with the underlying program/software. Most user interfaces have the ability to take-in information from the user and also to provide visual or audible information back to the user.

In the *real* world, an interface often has physical interactive components. For example, there are two obvious ways to interact with (or *interface* with) our bank account. We may go up to a teller at the bank and perform some transactions, or we may use an ATM.



In the *virtual* world, we may interact with our bank account electronically by using a web browser, or phone app or dedicated stand-alone software from the bank.



In this case, the interaction is through software menus, buttons, text fields, lists, etc..

In this course, we will consider software-based user interfaces such as those shown above. We will concentrate on stand-alone applications that do not require the use of a browser.

Up until this point, our programs/applications did not really have any interactive user interface. That is, we defined classes, wrote programs and then simply ran them inside of the IDE that we were using. The results of our program were displayed as graphics (in the case of Processing) or as text output (as in the case of the JAVA console window).

When writing programs that bring up a main interactive window (or those that run on a phone or in a browser), it is important to understand that more is "going on behind the scenes". That is, when we use an ATM machine, there is actually quite a bit "going on" in the way that our actions at the machine affect the current state of the bank and of our bank account. For example, the bank account changes, transaction logs are updated and security/error-checking is taking place. The ATM machine was simply making use of these underlying entities (i.e., the **bank** and the **account**) in order to complete the transaction. It is necessary at this point to bring up some definitions:

*The **model** of an application consists of all classes that represent the "business logic" part of the application ... the underlying system on which a user interface is attached.*

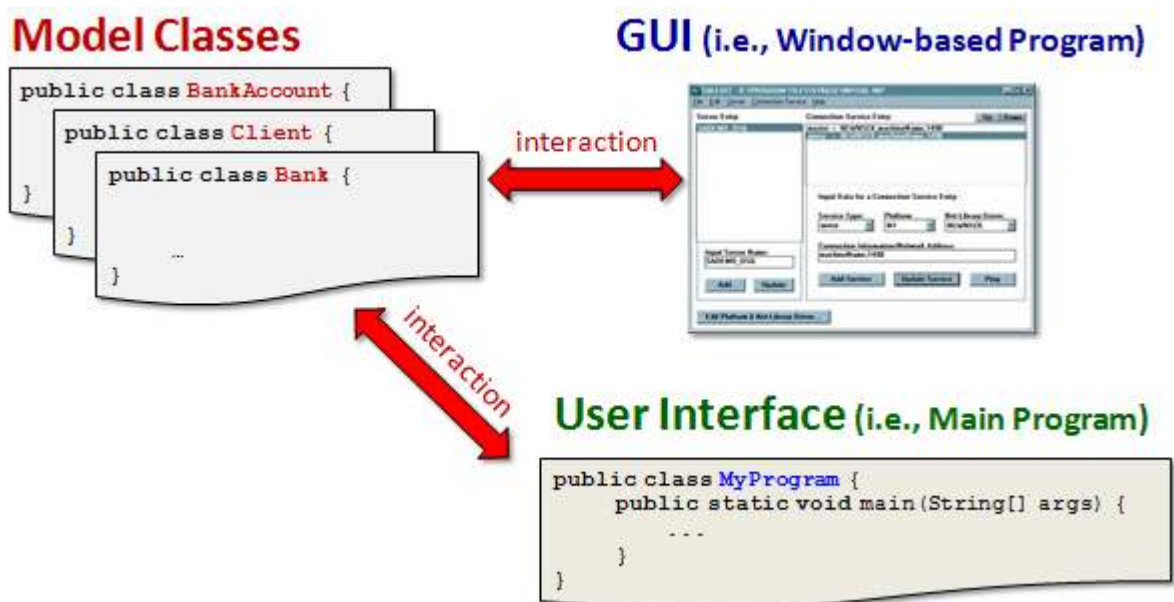
The model is always developed separately from the user interface. In fact, it should not assume any knowledge about the user interface at all (e.g., model classes should not assume that `System.out.println()` is available).

*The **user interface** is the part of the application that is attached to the model which handles interaction with the user and does NOT deal with the business logic.*

The user interface always makes use of the model classes and often causes the model to change according to user interaction. The changes to the model are often reflected back (visually) on the user interface as a form of immediate feedback.

*A **graphical user interface (GUI)** is a user interface that makes use of one or more windows to interact with the user.*

A GUI is often preferred over text-based user interfaces because it is more natural ... more like real-world applications that we are used to using. Imagine, for example, if the internet was only text-based with no buttons, text fields, drop-down lists, images, etc..



So, it is important to understand that there should always be a separation in your code between the *model* classes and the *user interface* classes. That will allow you to share the same model classes with different interfaces.

In JAVA, all of our main windows for our applications will be instances of the **JFrame** class. A **JFrame** will be the main starting place for our user interface-based programs. The JFrame window will contain window *components* that will allow us to interact with the user, such as buttons, text fields, lists etc...

Example:

The following code creates a simple window in JAVA. You can use this program as a template for all of your window-based applications:

```
import javax.swing.JFrame;

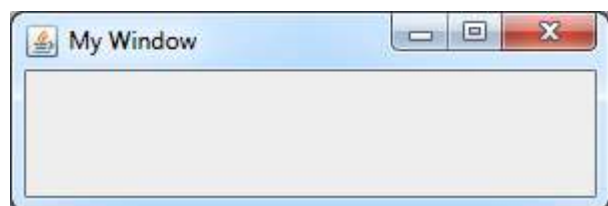
public class MyApplication extends JFrame {

    public MyApplication(String title) {
        super(title); // Set title of window
        setDefaultCloseOperation(EXIT_ON_CLOSE); // allow window to close
        setSize(300, 100); // Set size of window
    }

    public static void main(String[] args) {
        MyApplication frame;

        frame = new MyApplication("My Window"); // Create window
        frame.setVisible(true); // Show window
    }
}
```

Although this code for bringing up a new window can appear anywhere, we typically designate a whole class to represent the window (that is, the JAVA application). This also helps to separate the model and the user interface. So here are the steps involved with creating your own JAVA application that uses a main window (frame):



1. Create a new class (separate from model classes) to represent your application. In this case we created a **MyApplication.java** file to represent our window-based application.
2. Have this class extend **JFrame**. Make sure to *import* the **javax.swing.JFrame** package. This will allow us to inherit all of the JFrame's attributes and methods.
3. Create a constructor that sets the window title (specified as a parameter). In the constructor, set the size of the window using JFrame's **setSize()** method by specifying the width and height of the window (in pixels ... includes the frame around the window). If you do not set the size, the window will show up so small that you will only see part of the title bar.

4. Include a **main()** method as a starting point for the application that calls your constructor to make the window and then call the frame's **setVisible()** method with a value of **true** to make the window appear. When frames are created, by default they do not appear on the screen unless we call this method with a value of **true**.

Note as well that we specified for the application to **EXIT_ON_CLOSE**. This is necessary since we want the application to stop running when the window is closed. This is typical behavior for all applications that run under windowing operating system environments. What other choices do we have when the window is closed ? b We could have used:

```
// window is not closed
setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);

// window is hidden...the program keeps running
setDefaultCloseOperation(HIDE_ON_CLOSE);

// window is hidden and disposed of (more later)
setDefaultCloseOperation(DISPOSE_ON_CLOSE);
```

To test the application, just compile and run it as you normally do.

5.2 Components and Containers

In order for a window to be useful, it must contain various components.

*A **window component** is an object with a visual representation that is placed on a window and usually allows the user to interact with it.*

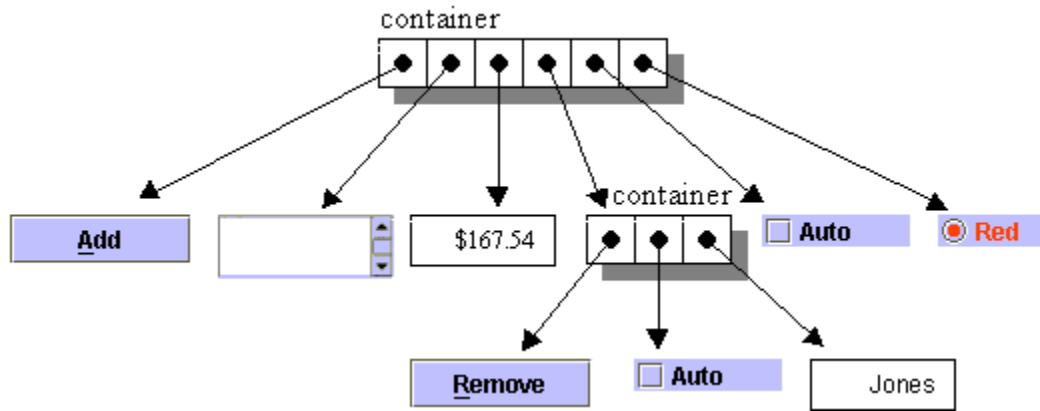
Typical window components are things such as buttons, text fields, drop down lists, scroll bars, tabbed panes, menus, etc..

Components may also be grouped together, much like adding elements to an array. In fact, most components in JAVA can contain other components as their sub-parts. This brings up the notion of a **Container**.

*A **container** is an object that contains components and/or other containers.*

The most easily understood container in JAVA is the **JFrame**, since windows generally contain many components on them. However, as you will see, there are other useful containers such as a **JPanel**. Containers are actually *components* as well. That is, you can have containers which contain other containers.

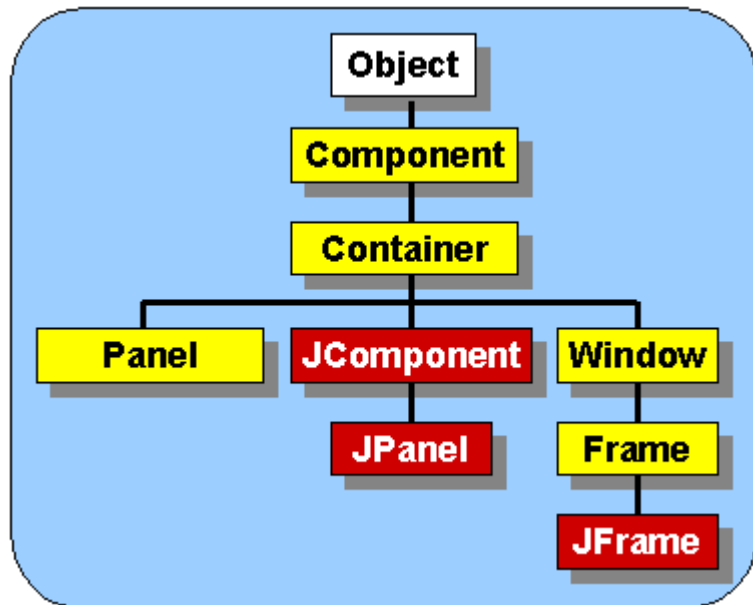
This allows for arrangements such as a **JFrame** containing two **JPanels**, each of which contains two **JLabels** etc.. So, a container of components is conceptually "like" an array of objects:



In the original versions of JAVA, all components were stored in the **java.awt** package. This package, and its sub-packages were known as the **Abstract Windowing Toolkit (AWT)**. Soon after, the JAVA guys developed a more platform-independent version in the **javax.swing** package. This package, and its sub-packages were known as the **Swing** packages. We will be making use of these newer **Swing** components.

Here is just a portion of the component hierarchy →

The **red** classes are classes in the swing packages, while the **yellow** ones are classes from the AWT packages:



Notice that a **JFrame** is a **Window** as well as a **Container**. **JPanels** are also **Containers** in that they too can contain many components.

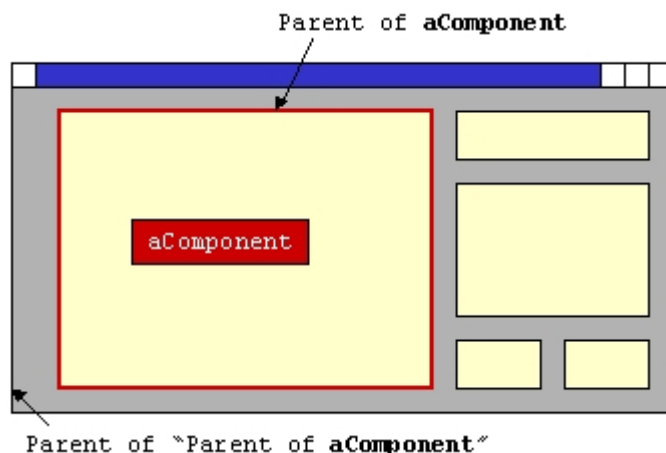
In fact, **Containers** themselves are **Components** and all **JComponents** are also **Containers**. So, in Swing, everything is a **Container** ... even a button!

All components actually keep pointers to their **parent** container (i.e., the component that contains it). Parents of nested components are stored recursively:

```

Component    aComponent;
Container    parent;
Container    parentOfParent;

aComponent = ... ;
parent = c.getParent();
parentOfParent = p.getParent();
    
```



Containers themselves keep pointers to their components (i.e., an array) and we can access these using the container's **getComponent()** and **getComponents()** methods:

```
Container    aParent = ... ;
Component    c1 = aParent.getComponent(0);
Component    c2 = aParent.getComponent(1);
Component    c3 = aParent.getComponent(2);

Component[]  c = aParent.getComponents(); // get them all
```

One of the most commonly used containers is called a **JPanel**:

*A **JPanel** is a frameless area on a window that usually contains a laid-out arrangement of other components.*

Think of a panel as a bulletin board that you can fill with components and then you can place the bulletin board anywhere as a component itself.



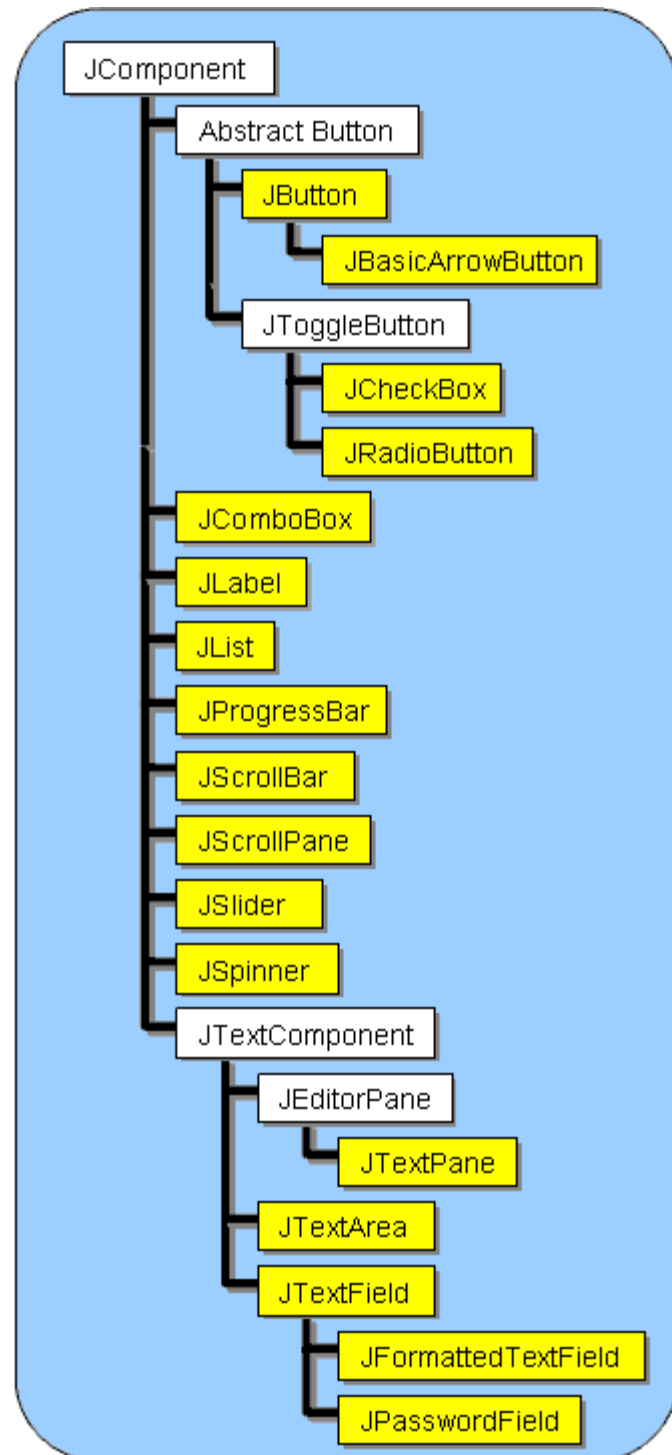
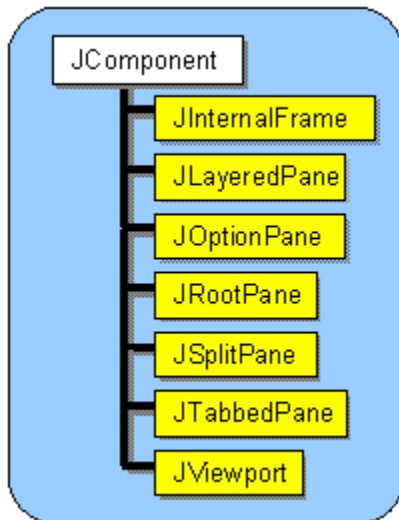
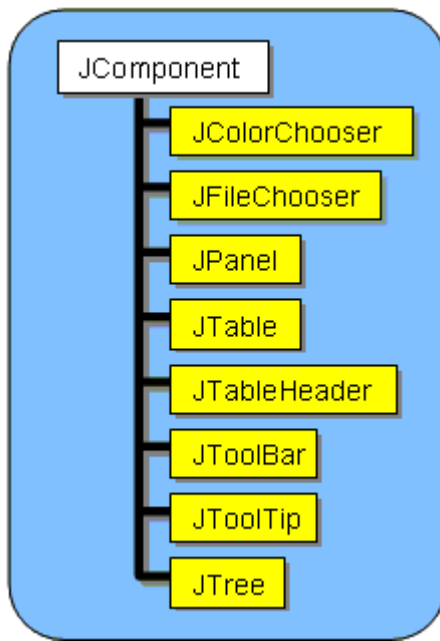
All **JFrames** have a **JPanel** at the top level to which all of our window components added. For simple "single-panel" windows, we can simply access this panel by sending the **getContentPane()** method to our **JFrame**, and then call **add()** to put components onto it:

```
Component    c1 = ...;
Component    c2 = ...;
Component    c3 = ...;

JFrame frame = new JFrame("MyApplication");
frame.getContentPane().add(c1);
frame.getContentPane().add(c2);
frame.getContentPane().add(c3);
```

Note that the component hierarchy diagram shown earlier does not list all available components. We can expand the **JComponent** hierarchy in more detail.

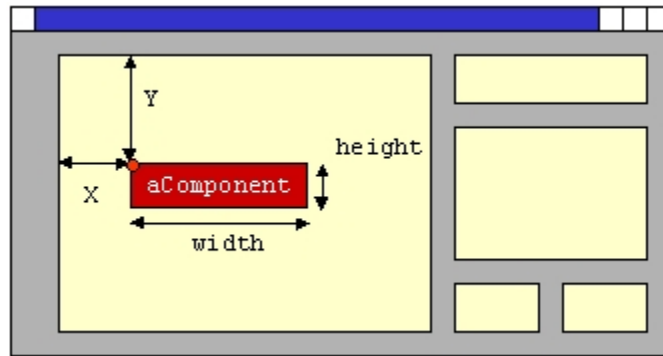
The three pictures on the next page (note that there was too much to show on one picture so it has been split into multiple pictures) show just some of the **JComponent** subclasses that represent commonly used components (shown in yellow) that are placed onto windows. Notice that all these **JComponents** start with a "J". Also notice that there are different kinds of buttons and text-based components.



There are even more subclasses ... those dealing with menus will be shown later.

You will get to know more about these components as we use them in our examples. For now, it is a good idea to understand the common properties and functionality that all components have.

All **JComponents** have an **(x, y)** location as well as a **width** and **height**. The location is the (x,y) coordinate of the top/left of the component. It is a coordinate within the coordinate system defined by the top left corner of the parent container (which is often a **JPanel**):



We can access or modify these values for the component at any time using the methods shown in the code below:

```

JComponent    c = ... ;

// ask a component for its location, its width and its height
int x = c.getX();
int y = c.getY();
int w = c.getWidth();
int h = c.getHeight();

// change a component's location
c.setLocation(new Point(100, 200));

// change a component's width and height
c.setSize(100, 50);

```

There is a problem with changing sizes and locations of components by default. JAVA has "Layout Managers" (more on this later) that automatically compute component locations and sizes. If we decide to use layout managers, we can "suggest" sizes for our components using the following methods:

```

c.setMaximumSize(new Dimension(width, height));
c.setMinimumSize(new Dimension(width, height));
c.setPreferredSize(new Dimension(width, height));

```

In addition to setting the size and dimensions of our components, we can also set the background and foreground colors of the component. The background color is the color that fills in the background of the component which is behind any text that appears on the component. The foreground color is the text color. Here, for example is a window with 4 JButtons on it, each with different background and foreground colors:



The code for setting the color of a component is simple:

```
JComponent    c = ... ;

c.setBackground(Color.red);
c.setForeground(Color.black);
```

There are a few "standards" colors definitions. Here are some of them:

```
Color.black    Color.green    Color.pink
Color.blue     Color.lightGray Color.red
Color.cyan     Color.magenta   Color.white
Color.darkGray Color.orange    Color.yellow
Color.gray
```

To use the above constants, you need to import the **Color** class which is located within the **java.awt** package using: **import java.awt.Color;**

However, these are not usually the best colors to use for a nice user interface, as they may "clash", resulting in an "ugly" user interface. You will usually want to define your own colors. The **Color** class provides constructors that allow you to create your own colors by specifying the amount of Red, Green and Blue that makes up the color. This can be done by either using integers (between 0 and 255) or floats (between 0.0 and 1.0) to represent a percentage of RGB values:

```
new Color(int r, int g, int b);
new Color(float r, float g, float b);
```

Here is an example:

```
JComponent    c = ... ;

c.setBackground(new Color(100, 50, 0)); // brown
c.setForeground(new Color(70, 0, 70));  // dark purple
```

In addition to setting the color, you may also choose the Font type (i.e., type of text) and Font size that appears on your component (e.g., button, text field, etc..). The **setFont()** method is used to do this. You need to supply a **Font** object.

```
JComponent    c = ... ;
Font           f = ...;

c.setFont(f);
```

You can create a **Font** object by calling a constructor in the **Font** class (located in the **java.awt** package). Here is one constructor that requires you to supply the name of the font, the style and the point size:

```
f = new Font(String name, int style, int size);
```

So, as an example, here is what you could write:

```
c.setFont(new Font("SansSerif", Font.BOLD, 12));
```

For the font **name**, you can supply any font that your system has available such as:

```
"Times", "Arial", "SansSerif" or "Courier"
```

To get a list of all fonts available on your computer, you can run the following code:

```
java.awt.GraphicsEnvironment    ge;
java.awt.Font[]                fonts;

ge = java.awt.GraphicsEnvironment.getLocalGraphicsEnvironment();
fonts = ge.getAllFonts();
for(int i=0; i<fonts.length; i++) {
    System.out.println(fonts[i].getName());
}
```

Here are possible **styles** (notice that we can "OR" them together using JAVA's bitwise OR operator |):

```
Font.BOLD, Font.ITALIC, Font.PLAIN, Font.BOLD|Font.ITALIC
```

Here is an example showing buttons with various font names, style and size. The rows show font sizes 8, 12, 18 and 18, respectively. The first 3 rows have PLAIN style while the last row had BOLD style. The columns (from left to right) show font types: **Script MT Bold**, **Arial Narrow**, **Courier** and **Times New Roman**, respectively:

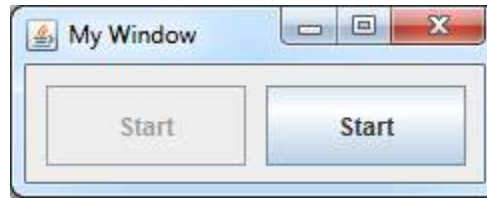


In addition to these visual parameters, we can also enable and disable components. A component that is disabled is "grayed out" and not able to be controlled by the user. A disabled button, for example, cannot be pressed. A disabled text field, for example, does not allow the user to type text into it.

We can enable and disable components at any time in our program by using the **setEnabled()** method as follows:

```
JComponent    c = ... ;

c.setEnabled(false);
...
c.setEnabled(true);
```

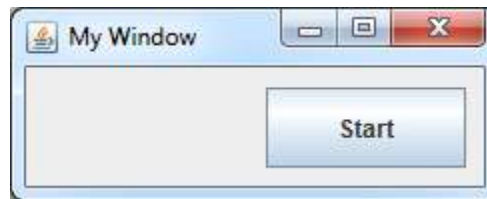


By default, all components are enabled and will remain that way unless you disable them.

Components can actually also be completely hidden from view. A **setVisible()** method is used to show or hide a component:

```
JComponent    c = ... ;

c.setVisible(false);
...
c.setVisible(true);
```



By default, all newly created components are visible (i.e., not hidden). However, **JFrames** are NOT automatically made visible ... you MUST do **setVisible(true)** to have your window appear.

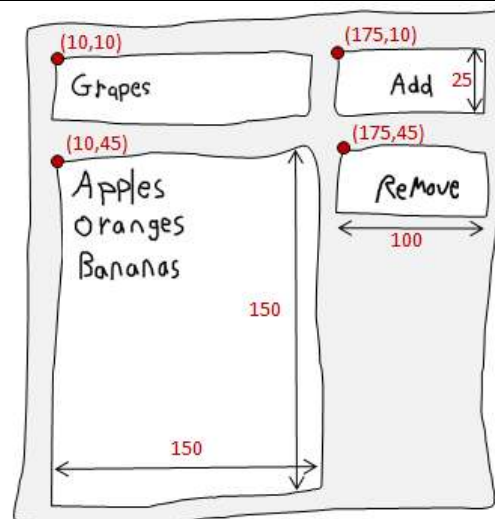
There are many more attributes that we can set for components and we will investigate some more of them throughout the course.

Example:

Consider writing a program that creates the window shown in this rough diagram →

The top/left component is a **JTextField**. The bottom/left component is a **JList** and the right two components are **JButtons** to add and remove items to the list.

It is always a good idea to draw out a rough version of your user interface, identifying the top/left corner or all components as well as their dimensions and the width or all margins around the window border and between the components.



To write the code, we begin with a basic class definition. Notice the line that says **setLayout(null)**. This allows us to manually place the components wherever we want to on the screen. Notice as well that we set the size of the window according to the dimensions in the picture with an extra 10 pixels wide to account for the window frame and an extra 25 pixels

high to account for the title bar of the window. We also made use of the **setResizable(false)** so that the window cannot be resized after we create it.

```
import javax.swing.*;

public class FruitListApp extends JFrame {
    public FruitListApp(String name) {
        super(name);

        // Choose to lay out components manually
        getContentPane().setLayout(null);

        // ... Add components here (see below) ...

        // Set program to stop when window closed
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(290, 230); // manually computed sizes
        setResizable(false);
    }

    public static void main(String[] args) {
        JFrame frame = new FruitListApp("My Fruit List");
        frame.setVisible(true);
    }
}
```

Now, we need to place our components on the window. Just insert the following code into the middle of the above constructor:

```
// Add the text field
JTextField newItemField = new JTextField("Grapes");
newItemField.setLocation(10,10);
newItemField.setSize(150,25);
getContentPane().add(newItemField);

// Add the ADD button
JButton addButton = new JButton("Add");
addButton.setLocation(175, 10);
addButton.setSize(100,25);
getContentPane().add(addButton);

// Add the REMOVE button
JButton removeButton = new JButton("Remove");
removeButton.setLocation(175,45);
removeButton.setSize(100,25);
getContentPane().add(removeButton);

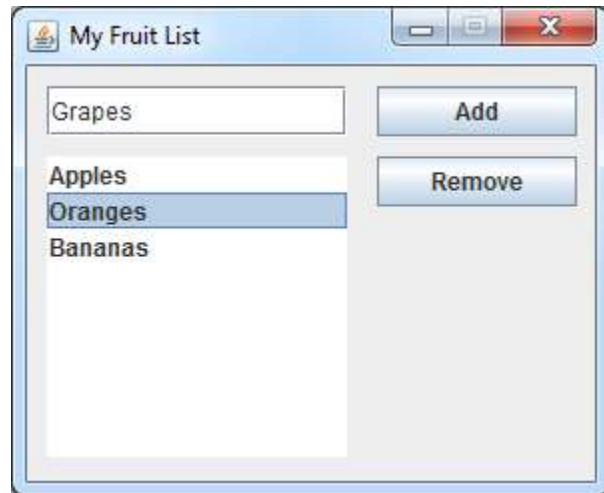
// Add the JList
String[] fruits = {"Apples", "Oranges", "Bananas"};
JList fruitList = new JList(fruits);
fruitList.setLocation(10,45);
fruitList.setSize(150,150);
getContentPane().add(fruitList);
```

Here is what the window will look like when we run the code →

You will notice that the text supplied in the **JTextField** constructor is the initial text that will appear in the text field when the window opens.

Also, the text supplied for the **JButtons** is the text that will appear on the button itself.

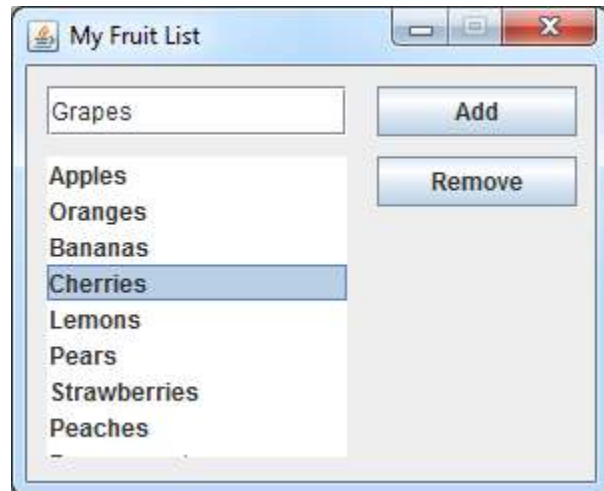
Lastly, notice that we can supply a list of items to place in the **JList** initially as an array of **String** objects. In fact, we can supply any objects for this list, but for now we will simply use Strings.



The code above adds a list with 3 items in it initially. If, however, the list had many items in it, the items cannot all be shown. For example, here is what the list would look like if we added these fruits:

```
String[] fruits = {
    "Apples", "Oranges", "Bananas",
    "Cherries", "Lemons", "Pears",
    "Strawberries", "Peaches",
    "Pomegranates", "Nectarines",
    "Apricots"};
```

Notice that the list cannot display more than 8 items due to its small size. In such a situation, we need scroll bars on our list so that the user can scroll to find particular items.



JAVA has a **JScrollPane** class that allows us to place components within it such that scroll bars are automatically placed in a way that allows scrolling in both the horizontal and vertical directions.

To use the scroll pane, we call the **JScrollPane** constructor with 3 parameters. The first parameter is the **JList** that we want to be able to scroll through (e.g., **fruitList** in our example). The next two parameters are the scroll policies for the vertical and horizontal scroll bars. For each scroll bar we must choose one of the following three policies for the vertical scroll bar:

```
ScrollPaneConstants.VERTICAL_SCROLLBAR_NEVER
ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS
ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED
```

... and choose one of these for the horizontal scroll bar:

```
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED
```

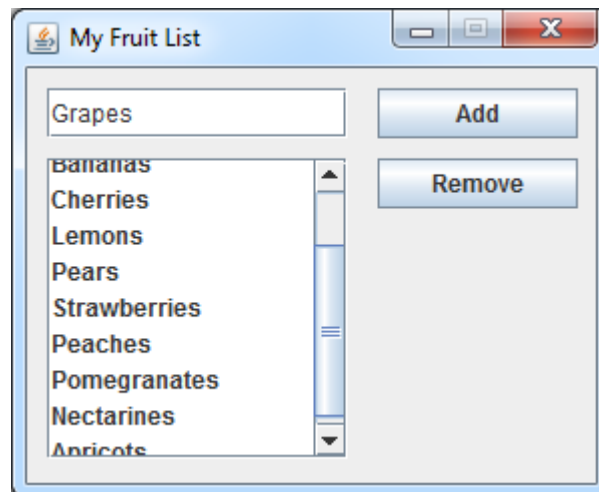
The `ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED` option will show no scroll bar unless one is needed.

That means, if there are not enough items in the list to require scrolling, the vertical scroll bar will not appear. Similarly, for horizontal scrolling, if all our items are short strings, the `HORIZONTAL_SCROLLBAR_AS_NEEDED` option will not show the scroll bar unless a longer item appeared in the list that cannot fit horizontally in the list.

As a result here is the new code:

```
String[] fruits = {"Apples", "Oranges", "Bananas", "Cherries",  
                  "Lemons", "Pears", "Strawberries", "Peaches",  
                  "Pomegranates", "Nectarines", "Apricots"};  
JList fruitList = new JList(fruits);  
  
JScrollPane scrollPane = new JScrollPane(fruitList,  
    ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,  
    ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);  
  
scrollPane.setLocation(10,45);  
scrollPane.setSize(150,150);  
getContentPane().add(scrollPane);
```

And here is the resulting window:

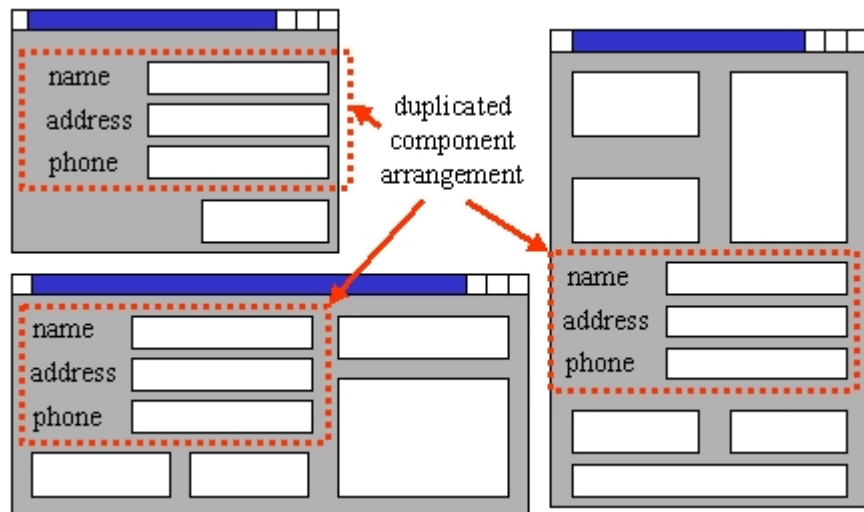


5.3 Grouping Components Together

It is a very good idea to keep your window components organized. It is often the case that an arrangement of components may be similar (or duplicated) within different windows.

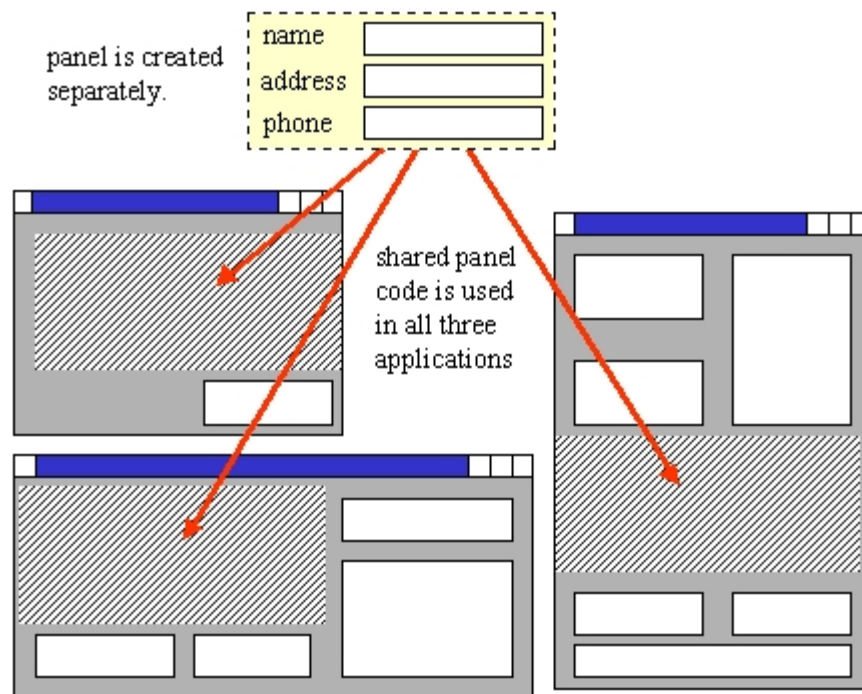
For example, an application may require a name, address and phone number to be entered at different times in different windows →

It is a good idea to share component layouts among the similar windows within an application so that the amount of code you write is reduced.



To do this, we often lay out components onto a **JPanel** and then place the **JPanel** on our window. We can place the created panel on many different windows with one line of code ... this can dramatically reduce the amount of GUI code that you have to write.

So, you will often want to create separate **JPanels** to contain groups of components so that you can move them around (as a group) to different parts of a window or even be shared between different windows. The code to do this simply involves creating our **JPanel** with its appropriate component arrangement and then adding the **JPanel** to the **JFrame**.



On the next page is an example that takes two **JPanels** with 2 components each on them and then adds them to a **JFrame**. Note that some coding details have been left out purposely:

```

Component    c1, c2, c3, c4, c5;
JPanel       p1, p2;

/* ... code omitted for creating the components and panels ... */

JFrame frame = new JFrame("MyApplication");
p1.add(c1);
p1.add(c2);
p2.add(c3);
p2.add(c4);
frame.getContentPane().add(p1);
frame.getContentPane().add(p2);
frame.getContentPane().add(c5);

```

Notice how components `c1` and `c2` are added to panel `p1` and then `c3` and `c4` are added to panel `p2`. Then these panels are simply added to the frame's content pane as if they were simply components. The following example is more specific.

Example:

Consider another example in which a **JPanel** is used in more than one window. We will create a simple panel called **AddressPanel** that contains 5 labels and 5 text fields for allowing the user to enter a name and address as shown here →

The panel contains 5 **JTextField** objects that allow the user to fill-in an address. It also has 5 **JLabel** objects (which are simply pieces of text) to indicate the kind of data expected for each text field. Lastly, there is a nice border around the **JPanel** which has the title CONTACT ADDRESS. This panel will not be its own window. Instead, it will be a panel inside of two other windows that look as shown here. Notice that each application has the same kind of **AddressPanel**, except that the border's title varies.

We will need to compute the locations and sizes for each component. Note that the CONTACT ADDRESS is not a **JLabel** but is actually part of the panel's border, as you will soon see. Here are the dimensions →

To begin the code, we will want to define an **AddressPanel** class. It will be a special kind of **JPanel**, and so it should be a subclass of **JPanel**. The code should have the following framework:

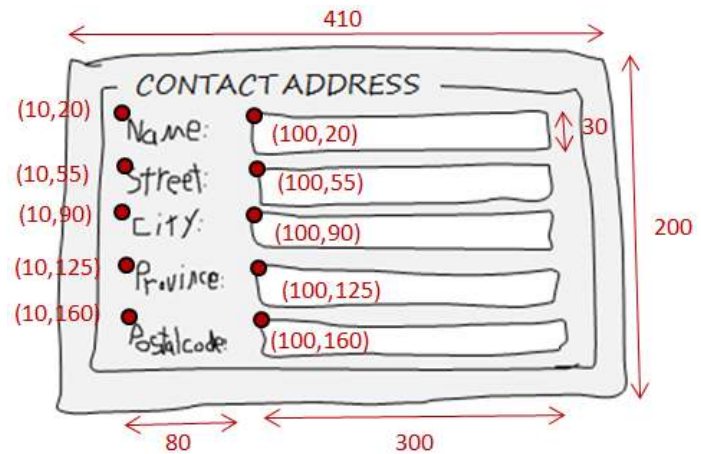
```
import javax.swing.*;

public class AddressPanel extends JPanel {
    public AddressPanel(String title) {
        // Choose to lay out components manually
        setLayout(null);

        // Make a border around the outside with the given title
        setBorder(BorderFactory.createTitledBorder(title));

        // ... Add components here (see below) ...

        setSize(410, 200);
    }
}
```



You may notice that the **AddressPanel** takes a constructor that allows the user to pass in a **String** to be the title on the border. This title is used in the call to **setBorder()** in order to make the appropriate border with that title. The components are then added to the window one-by-one as follows:

```
// Add the Name, Street, City, Province and PostalCode labels
JLabel aLabel = new JLabel("Name:");
aLabel.setLocation(10, 20);
aLabel.setSize(80, 30);
add(aLabel);

aLabel = new JLabel("Street:");
aLabel.setLocation(10, 55);
aLabel.setSize(80, 30);
add(aLabel);

aLabel = new JLabel("City:");
aLabel.setLocation(10, 90);
aLabel.setSize(80, 30);
add(aLabel);

aLabel = new JLabel("Province:");
aLabel.setLocation(10, 125);
aLabel.setSize(80, 30);
add(aLabel);
```

```

aLabel = new JLabel("Postal Code:");
aLabel.setLocation(10, 160);
aLabel.setSize(80, 30);
add(aLabel);

// Add the name textfield
JTextField nameField = new JTextField();
nameField.setLocation(100, 20);
nameField.setSize(300, 30);
add(nameField);

// Add the street textfield
JTextField streetField = new JTextField();
streetField.setLocation(100, 55);
streetField.setSize(300, 30);
add(streetField);

// Add the city textfield
JTextField cityField = new JTextField();
cityField.setLocation(100, 90);
cityField.setSize(300, 30);
add(cityField);

// Add the province textfield
JTextField provinceField = new JTextField();
provinceField.setLocation(100, 125);
provinceField.setSize(300, 30);
add(provinceField);

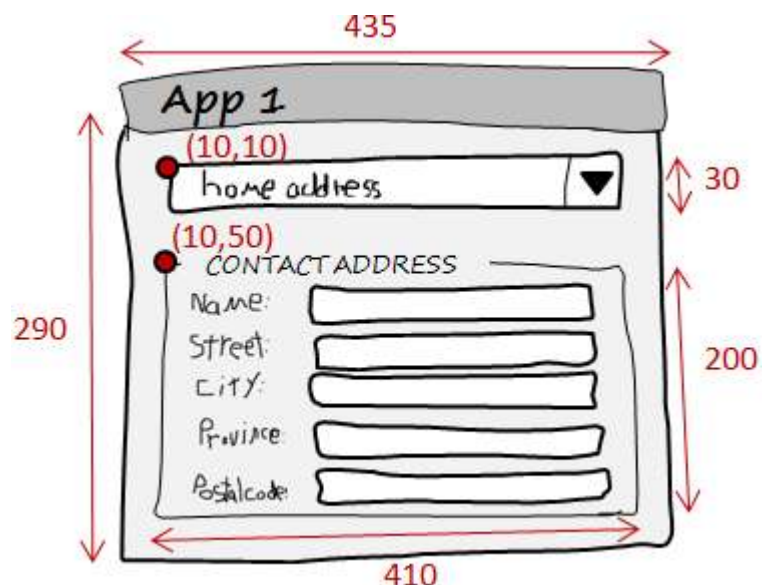
// Add the postal code textfield
JTextField postalField = new JTextField();
postalField.setLocation(100, 160);
postalField.setSize(300, 30);
add(postalField);

```

The **AddressPanel** itself is not a runnable application (i.e., there is no **main** method). So we will now make our App1 application to test it out. Here are the dimensions for the application →

Notice that the **AddressPanel** is now "treated" as a single component and is placed on the main window by specifying its location. We do not need to specify the size of the **AddressPanel**, since this was defined within the **AddressPanel** constructor and is fixed.

The topmost component is called a **JComboBox** and it represents what is known as a drop-down list. It is similar to a **JList**, except that only the selected item is shown and the remaining items can be shown by pressing the black arrow.



A **JComboBox** can be created by specifying an array of objects that are to appear in the list and passing this in as a parameter to the constructor as follows:

```
String[] addresses = {"Home Address", "Work Address", "Alternate Address"};
JComboBox addressBox1 = new JComboBox(addresses);
```

Here is the code to create the application. Take note of how the **AddressPanel** is now used as if it were a single, simple component:

```
import javax.swing.*.*;

public class OneApp extends JFrame {
    public OneApp(String name) {
        super(name);

        // Choose to lay out components manually
        getContentPane().setLayout(null);

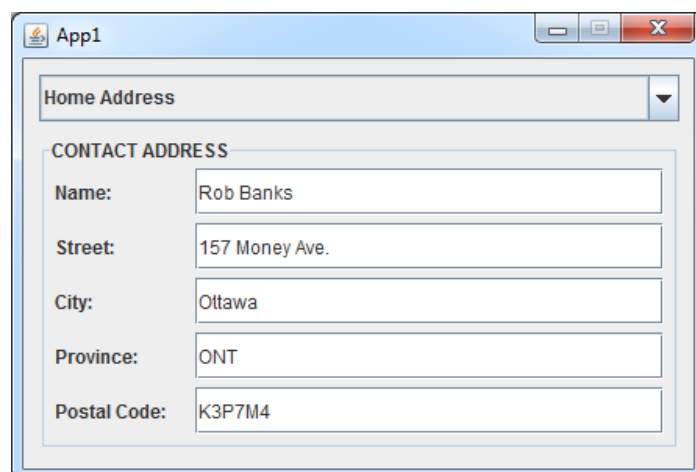
        // Add the drop-down list
        String[] addresses = {"Home Address", "Work Address",
                              "Alternate Address"};
        JComboBox addressBox1 = new JComboBox(addresses);
        addressBox1.setLocation(10,10);
        addressBox1.setSize(410,30);
        getContentPane().add(addressBox1);

        // Now add an AddressPanel
        AddressPanel myPanel = new AddressPanel("CONTACT ADDRESS");
        myPanel.setLocation(10,50);
        getContentPane().add(myPanel);

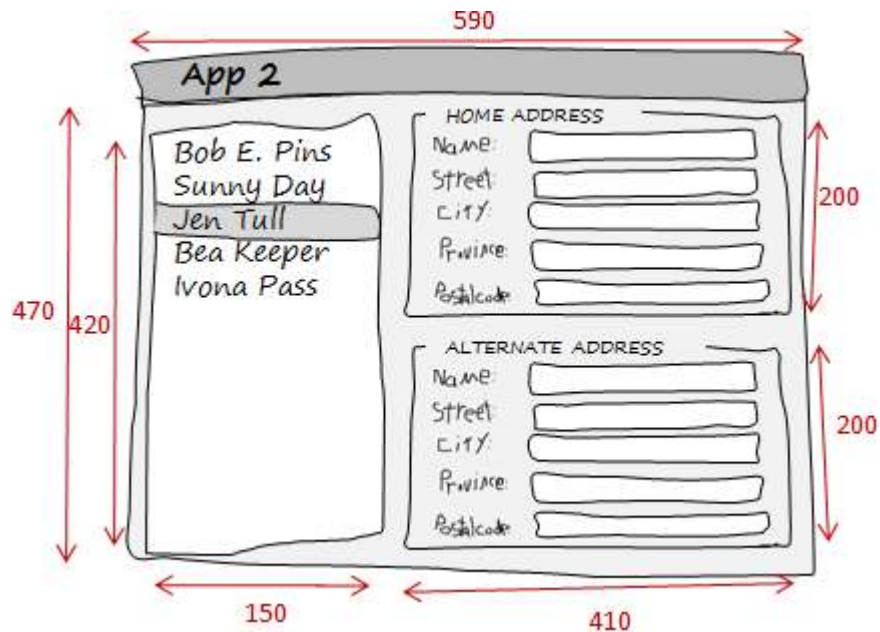
        // Set program to stop when window closed
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(435, 290); // manually computed sizes
        setResizable(false);
    }

    public static void main(String[] args) {
        JFrame frame = new OneApp("App1");
        frame.setVisible(true);
    }
}
```

And here is the finished product →



Here are the dimensions for the 2nd application:



Here is the code. Again, notice how the **AddressPanel** is used twice in the same window with a different title:

```
import javax.swing.*;
public class TwoApp extends JFrame {
    public TwoApp(String name) {
        super(name);

        // Choose to lay out components manually
        getContentPane().setLayout(null);

        // Add the list of names
        String[] names = {"Bob E. Pins", "Sunny Day", "Jen Tull",
            "Bea Keeper", "Ivona Pass"};
        JList aList = new JList(names);
        JScrollPane scrollPane = new JScrollPane(aList,
            ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
            ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        scrollPane.setLocation(10,10);
        scrollPane.setSize(150,420);
        getContentPane().add(scrollPane);

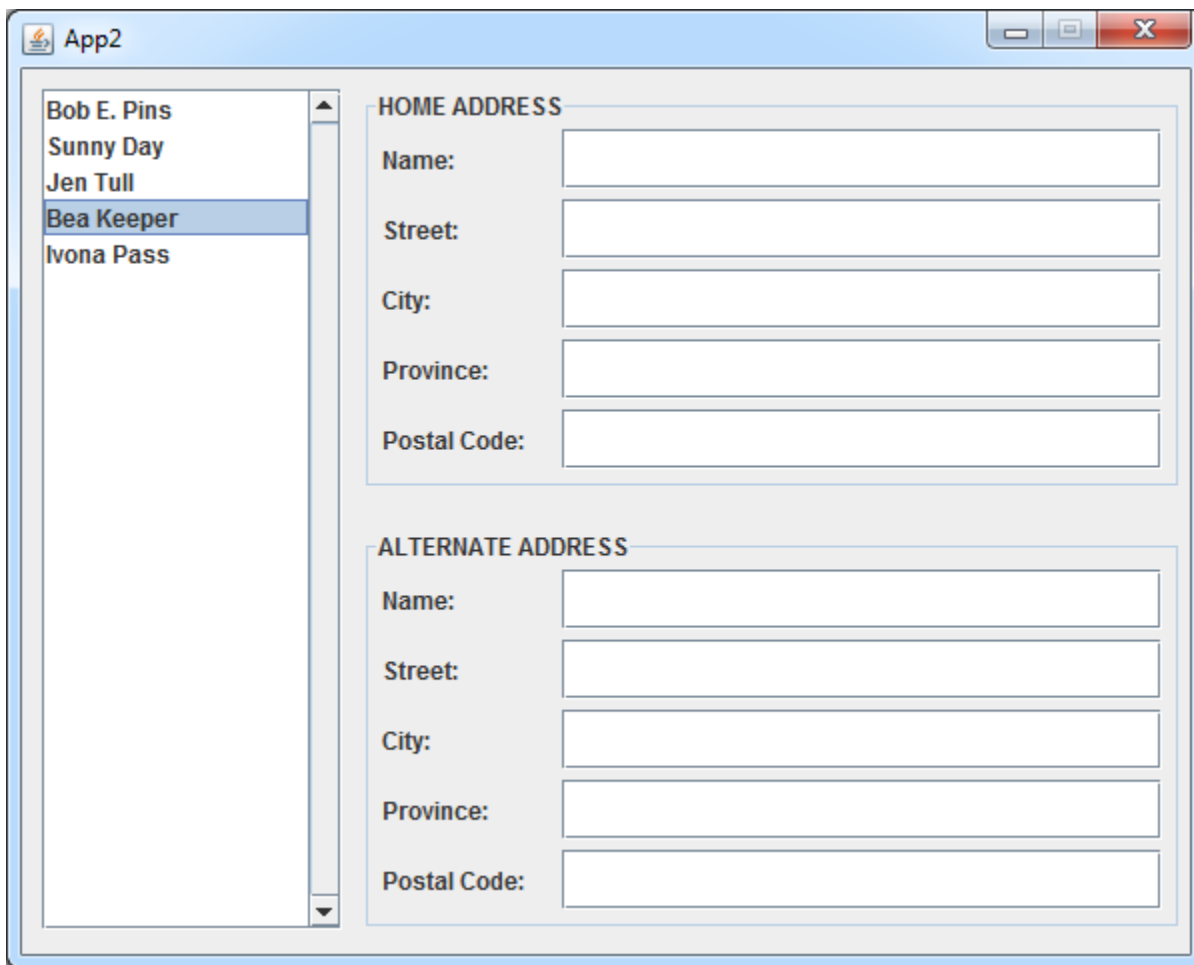
        // Now add an AddressPanel
        AddressPanel myPanel1 = new AddressPanel("HOME ADDRESS");
        myPanel1.setLocation(170,10);
        getContentPane().add(myPanel1);

        // Now add an another AddressPanel
        AddressPanel myPanel2 = new AddressPanel("ALTERNATE ADDRESS");
        myPanel2.setLocation(170,230);
        getContentPane().add(myPanel2);
    }
}
```

```
// Set program to stop when window closed
setDefaultCloseOperation(EXIT_ON_CLOSE);
setSize(590, 470); // manually computed sizes
setResizable(false);
}

public static void main(String[] args) {
    JFrame frame = new TwoApp("App2");
    frame.setVisible(true);
}
}
```

And ... the result:



5.4 Event Handling

Now that we understand the basics of laying out various components on our graphical user interfaces, we need to discuss how to allow the user to interact with the window and make things happen.

Recall these definitions from the previous course:

*An **event** is something that happens in the program based on some kind of triggering input which is typically caused (i.e., **generated**) by user interaction such as pressing a key on the keyboard, moving the mouse, or pressing a mouse button.*

*An **event handler** is a procedure that specifies the code to be executed when a specific type of event occurs in the program.*

In the previous course, you may have written event handlers for handling events for when the mouse button was pressed, released or clicked as well as when the mouse was moved or dragged or when a key on the keyboard was pressed, released or clicked. As an example you may have written event handlers like this:

```
void mousePressed() {
    if (dist(x,y,mouseX,mouseY) < RADIUS)
        grabbed = true;
}

void mouseReleased() {
    if (grabbed) {
        direction = atan2(mouseY - pmouseY, mouseX - pmouseX);
        speed = int(dist(mouseX, mouseY, pmouseX, pmouseY));
    }
    grabbed = false;
}
```

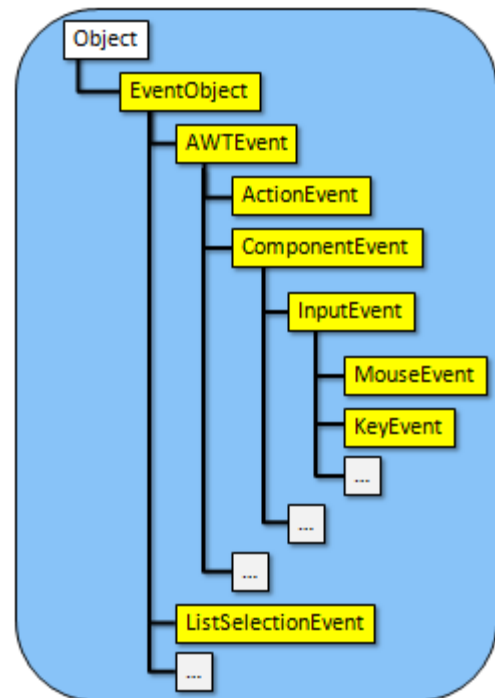
When writing GUIs we need to write specific event handlers in order for our application to respond to button clicks, allow selecting items from a list, allow typing into a text field etc...

There are two terms that we need to define:

*The **source** of an event is the component for which the event was generated (i.e., when handling button clicks, the **Button** is the **source**).*

*A **listener** is an event handler (i.e., also known as a **callback** procedure).*

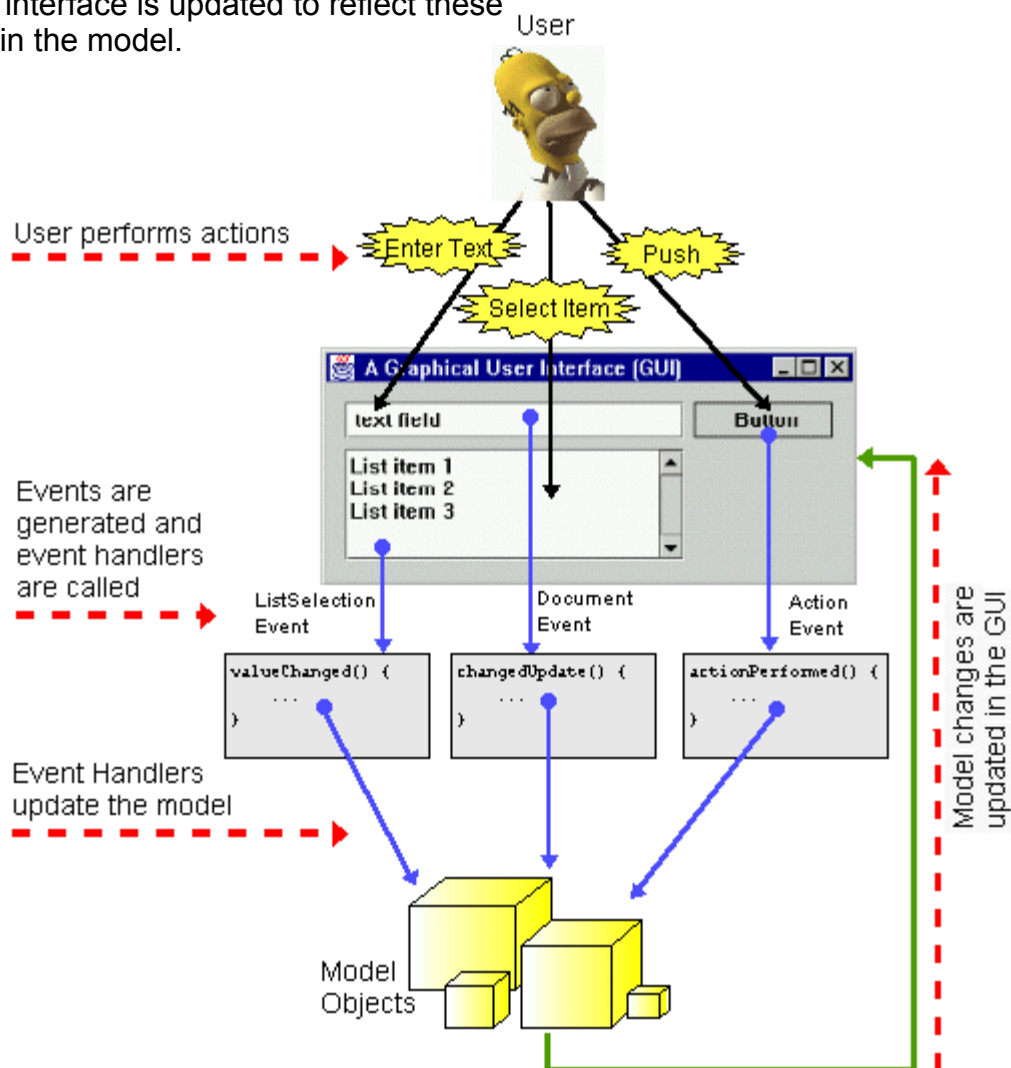
In JAVA, all **Events** are represented by a distinct class. There are many kinds of events, each having its own unique class. Here is a partial hierarchy showing some of the events that we will use in the course →



You do not need to memorize these, but you will become familiar with them during the course.

These events are generated when the user interacts with the user interface as follows:

1. The user causes an event by clicking a button, entering text, selecting a list item etc..
2. JAVA calls the appropriate event handler.
3. The event handling code changes the model in some way.
4. The user interface is updated to reflect these changes in the model.



Therefore, to perform event handling in JAVA, you must first identify the event that you want to handle. Then you need to write the appropriate event handlers. For each event, there is a corresponding *interface* in JAVA with a list of methods that you can write in order to handle the appropriate event in a meaningful way.

Below is a table of commonly-used events along with their Listener interfaces. This list basically tells you which methods need to be written in order to handle the kind of event that you are interested in. For a more complete description of these (and other) events, listeners and their methods, see the JAVA API specifications.

Event	Interface to Implement
ActionEvent - generated when button pressed, menu item selected, enter key pressed in a text field or from a timer event	<pre>public interface ActionListener { public void actionPerformed(ActionEvent e); }</pre>
DocumentEvent - generated when changes have been made to a text document such as insertion, removal in an editor	<pre>public interface DocumentListener { public void changedUpdate(DocumentEvent e); public void insertUpdate(DocumentEvent e); public void removeUpdate(DocumentEvent e); }</pre>
ListSelectionEvent - generated when selecting (i.e., click or double click) a list item	<pre>public interface ListSelectionListener { public void valueChanged(ListSelectionEvent e); }</pre>
WindowEvent - generated when open/close, activate/deactivate, iconify/deiconify a window	<pre>public interface WindowListener { public void windowOpened(WindowEvent e); public void windowClosed(WindowEvent e); public void windowClosing(WindowEvent e); public void windowActivated(WindowEvent e); public void windowDeactivated(WindowEvent e); public void windowIconified(WindowEvent e); public void windowDeiconified(WindowEvent e); }</pre>
KeyEvent - generated when pressing and/or releasing a key while within a component	<pre>public interface KeyListener { public void keyPressed(KeyEvent e); public void keyReleased(KeyEvent e); public void keyTyped(KeyEvent e); }</pre>
MouseEvent - generated when pressing/releasing/clicking a mouse button, moving a mouse onto or away from a component	<pre>public interface MouseListener { public void mouseClicked(MouseEvent e); public void mouseEntered(MouseEvent e); public void mouseExited(MouseEvent e); public void mousePressed(MouseEvent e); public void mouseReleased(MouseEvent e); }</pre>
MouseEvent - generated when moving mouse within a component while button is up or down	<pre>public interface MouseMotionListener { public void mouseDragged(MouseEvent e); public void mouseMoved(MouseEvent e); }</pre>

So what does all of this mean ? It means, for example, that if you want to handle a button press in your program, you need to write an **actionPerformed()** method:

```
public void actionPerformed(ActionEvent e) {
    //Do what needs to be done when the button is clicked
}
```

If you want to have something happen when the user presses a particular key on the keyboard, you need to write a **keyPressed()** method:

```
public void keyPressed(KeyEvent e) {
    //Do what needs to be done when a key is pressed
}
```

Once we decide which events we want to handle and then write our event handlers, we then need to **register** the event handler. This is like "plugging-in" the event handler to our window. In general, many applications can *listen for* events on the same component. So when the component event is generated, JAVA must inform everyone who is listening. We must therefore tell the component that we are listening for (or waiting for) an event. If we do not tell the component, it will not notify us when the event occurs (i.e., it will not call our event handler). So, when a component wants to signal/fire an event, it sends a specific message to all listener objects that have been registered (i.e., anybody who is "listening"). For every event, therefore, that we want to handle, we must not only write the listener (i.e., event handler) but also **register** that listener.



To help you understand this notion of registering, imagine signing up on a webpage somewhere to receive an email notification when some event occurs (e.g., when something goes on sale, or getting an email bill-statement at the end of the month). When we sign up, we are essentially registering for (or listening to) any updates that may occur as a result of the event.



To **register** for an event (i.e., enable it), we need to merely add the listener (i.e., your event handler) to the component by using an **addXXXListener()** method (where XXX depends on the type of event to be handled). Here are some examples:

```
aButton.addActionListener(anActionListener);
aJPanel.addMouseListener(aMouseListener);
aJFrame.addWindowListener(aWindowListener);
```

Here **anActionListener**, **aMouseListener** and **aWindowListener** can be instances of any class that implements the specific Listener interface.

So, for example, if you wanted to have your application handle a button press, you can make your application itself be the **ActionListener** as follows:

```

import java.awt.event.*; // Need this for ActionEvent and ActionListener
import javax.swing.*;    // Need this for JFrame and  JButton

public class SimpleEventTest extends JFrame implements ActionListener {
    public SimpleEventTest(String name) {
        super(name);

        getContentPane().setLayout(null);

        JButton aButton = new JButton("Press Me");
        aButton.setLocation(20,10);
        aButton.setSize(100, 30);
        getContentPane().add(aButton);

        // Plugin button event handler using THIS class as the listener
        // (i.e., tell JAVA to call the actionPerformed() in THIS class)
        aButton.addActionListener(this);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(250, 90);
    }
    // Must write this method now since SimpleEventTest
    // implements the ActionListener interface
    public void actionPerformed(ActionEvent e) {
        System.out.println("I have been pressed");
    }

    public static void main(String[] args) {
        JFrame frame = new SimpleEventTest("Making a Listener");
        frame.setVisible(true);
    }
}

```



Sometimes it is necessary to remove a listener (i.e., disable it). As we will see later, we often disable a listener while we are making changes to update the user interface so that additional events do not get generated automatically.

You can "unregister" from an event (i.e., disable the listener), by merely removing it using a **removeXXXListener()** method. Here are some examples:

```

aButton.removeActionListener(anActionListener);
aJPanel.removeMouseListener(aMouseListener);
aJFrame.removeWindowListener(aWindowListener);

```

It is important that you remove the listener that was previously added. Therefore, it is often necessary to store the listener in a variable so that it can be removed later:

```

ActionListener    buttonHandler = this;
...
aButton.addActionListener(buttonHandler);
...
aButton.removeActionListener(buttonHandler);
...

```

In some situations, we only want to handle one or two types of user interactions and therefore want to only write the event handlers needed to deal with that type of interaction. For example we may want to handle a **windowOpened()** event handler to do something when the window is first opened, but we may not care about when the window is closed, activated, iconified, etc..

Unfortunately, if we write code in the style as we did above by implementing the **WindowListener** interface, we would be forced to write 7 event handlers (since JAVA forces you to write ALL the methods defined in an interface that we implement):

```
public class WindowEventTest extends JFrame implements WindowListener {
    public WindowEventTest (String name) {
        ...
        // Plugin window event handler
        this.addWindowListener(this);
        ...
    }

    // Unfortunately, we now have to write 7 methods as follows,
    // (even though we really only want to write one):

    public void windowOpened(WindowEvent e) {
        System.out.println("Window has been opened");
    }
    public void windowClosed(WindowEvent e) { /* leave blank */ }
    public void windowClosing(WindowEvent e) { /* leave blank */ }
    public void windowActivated(WindowEvent e) { /* leave blank */ }
    public void windowDeactivated(WindowEvent e) { /* leave blank */ }
    public void windowIconified(WindowEvent e) { /* leave blank */ }
    public void windowDeiconified(WindowEvent e) { /* leave blank */ }

    public static void main(String[] args) { ... }
}
```

Writing these extra "empty" event handling methods is a lot of extra code writing that just wastes time and makes the code more confusing. It does seem a little silly to have to write 6 blank methods when we do not even want to handle these other kinds of events. The JAVA guys recognized this inconvenience and solved it using the notion of **Adapter** classes.

*An **adapter class** is a class that is used to implement an interface having a set of dummy methods.*

For each listener interface that has more than one method specified, there exists an adapter class with a corresponding name:

- **MouseListener** has **MouseAdapter**
- **MouseMotionListener** has **MouseMotionAdapter**
- **DocumentListener** has **DocumentAdapter**
- **WindowListener** has **WindowAdapter**
- ...and so on.



ActionListener and **ListSelectionListener** do NOT have an

adapter class since they only define one method each.

Here, for example is the **WindowAdapter** class:

```
public abstract class WindowAdapter implements WindowListener {
    public void windowOpened(WindowEvent e) {};
    public void windowClosed(WindowEvent e) {};
    public void windowClosing(WindowEvent e) {};
    public void windowActivated(WindowEvent e) {};
    public void windowDeactivated(WindowEvent e) {};
    public void windowIconified(WindowEvent e) {};
    public void windowDeiconified(WindowEvent e) {};
}
```

Adapter classes are provided for convenience sake to help us avoid writing empty methods. We can write subclasses that **extend** adapter classes, thereby inheriting the blank methods. Therefore, we would only need to write the event handlers that we are interested in, allowing the dummy methods to be inherited.

Unfortunately, looking at our previous **WindowEventTest** program, we cannot simply extend **WindowAdapter** since the code extends **JFrame** already and JAVA only allows us to extend a single class.

As a solution, we could create a new **inner class** just for the event handler and then use that:

```
import java.awt.event.*;    // Need this for WindowEvent and WindowListener
import javax.swing.*;      // Need this for JFrame

public class WindowEventTest2 extends JFrame {
    public WindowEventTest2 (String name) {
        super (name);

        // Plugin window event handler by creating a separate class
        // that extends WindowAdapter so that only one method needs
        // to be written.  This is called an "inner class", which
        // must have default access (e.g., not public nor private).
        class MyWindowHandler extends WindowAdapter {
            public void windowOpened(WindowEvent event) {
                System.out.println("Window has been opened");
            }
        }

        this.addWindowListener(new MyWindowHandler());

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(400, 300);
    }
    public static void main(String[] args) {
        JFrame frame = new WindowEventTest2("Inner Class Example");
        frame.setVisible(true);
    }
}
```

This is the first time that we created a class within another class explicitly.

An **inner class** is a class declared entirely within the body of another class or interface.

Interestingly, when you create an inner class, the JAVA compiler will create an additional class file (in this case it is called **WindowEventTest2\$1MyWindowHandler.class**). As you write more user interfaces, you will find many such class files created... each identified by a \$ character in the name.

To reduce clutter in your program, JAVA allows another shorter syntax for creating inner classes. It is a way of defining a class without specifying a name for the class. Here is the syntax:

```
import java.awt.event.*;    // Need this for WindowEvent and WindowListener
import javax.swing.*;      // Need this for JFrame

public class WindowEventTest3 extends JFrame {
    public WindowEventTest3 (String name) {
        super (name);

        // Plugin window event handler by creating an anonymous class
        // that extends WindowAdapter.
        this.addWindowListener(new WindowAdapter() {
            public void windowOpened(WindowEvent event) {
                System.out.println("Window has been opened");
            }
        });
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(250, 90);
    }
    public static void main(String[] args) {
        JFrame frame = new WindowEventTest3("Anonymous Class Example");
        frame.setVisible(true);
    }
}
```

This syntax actually creates and makes an instance of an inner class as a subclass of **WindowAdapter**. The class has no name, it is considered to be an **anonymous** class. This code actually creates an instance of the anonymous class and returns it to us. It is weird syntax. The .class file produced from this anonymous class will be titled: **WindowEventTest3\$1.class**

This idea of using anonymous classes is the simplest and most compact code for adding event handlers when only one method from a listener interface is needed. We will use adapter classes often in our examples.

Example:

How could we write a program that was able to distinguish between two different buttons on the window ?



Adding the buttons to the window is easy. To connect their event handlers, we have two choices. We can either:

- 1) write separate event handlers for each button, or
- 2) write a single event handler for both buttons.

In this example, we will choose the first option. Here is the code with separate event handlers:

```
import java.awt.event.*;    // Needed for ActionListener and(ActionEvent)
import javax.swing.*;      // Needed for JFrame and JButton

public class TwoButtonsApp extends JFrame {
    public TwoButtonsApp(String title) {
        super(title);
        getContentPane().setLayout(null);

        JButton button1 = new JButton("Press Me");
        button1.setLocation(20,10); button1.setSize(150, 30);
        getContentPane().add(button1);

        JButton button2 = new JButton("Don't Press Me");
        button2.setLocation(190,10); button2.setSize(150, 30);
        getContentPane().add(button2);

        // Add the first button's event handler
        button1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("That felt good!");
            }
        });

        // Add the second button's event handler
        button2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Ouch! Stop that!");
            }
        });

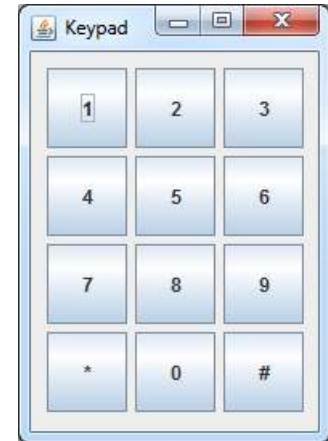
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(370,90);
    }

    public static void main(String args[]) {
        TwoButtonsApp frame = new TwoButtonsApp("Two Buttons");
        frame.setVisible(true);
    }
}
```

Example:

The previous example showed a good solution if we only have a small number of buttons. However, if we have more buttons, it is often good to have them share the same event handler. One way to do this is to have the **JFrame** implement **ActionListener** and then point all the buttons to it. We will create the following keypad using one event handler →

To do this, we will have the main application implement **ActionListener** and point all buttons to that same listener. The event handler will be written as a separate method (i.e., outside of the constructor), so we will need to store the buttons as instance variables (i.e., object attributes) so that we can access the buttons from both the constructor AND the event handler:



```
import java.awt.event.*;           // Needed for ActionListener and(ActionEvent)
import javax.swing.*;             // Needed for JFrame and JButton

public class MultipleButtonsApp extends JFrame implements ActionListener {
    // This stores all buttons
    JButton[][] buttons;

    public MultipleButtonsApp(String title) {
        super(title);
        getContentPane().setLayout(null);

        buttons = new JButton[4][3];
        String[] buttonLabels = {"1", "2", "3", "4", "5", "6", "7", "8", "9", "*", "0", "#"};
        for(int row=0; row<4; row++) {
            for (int col=0; col<3; col++) {
                buttons[row][col] = new JButton(buttonLabels[row*3+col]);
                buttons[row][col].setLocation(10+col*55, 10+row*55);
                buttons[row][col].setSize(50,50);
                buttons[row][col].addActionListener(this);
                getContentPane().add(buttons[row][col]);
            }
        }

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(195,275);
    }

    // This is the single event handler for all the buttons
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button " + e.getActionCommand() + " was pressed.");
    }

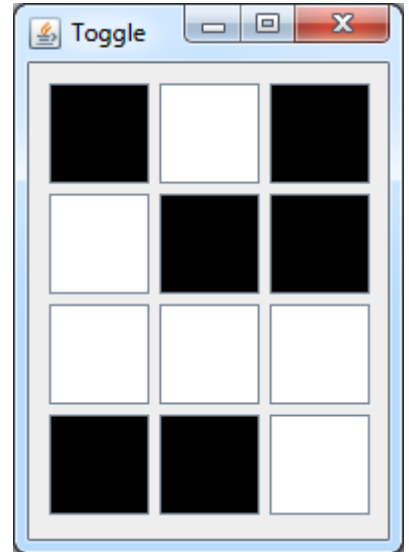
    public static void main(String args[]) {
        MultipleButtonsApp frame = new MultipleButtonsApp("Keypad");
        frame.setVisible(true);
    }
}
```

You may notice that we have called the **getActionCommand()** method for the **ActionEvent** that is passed in as a parameter in the event handler. This method retrieves the text from the button that was pressed.

Example:

Another way of determining the button that was pressed is to use the **getSource()** method for the **ActionEvent**. This will return the component that generated the event (i.e., the actual **JButton** object representing the button that was pressed). This is particularly useful if we have a grid of buttons that have no labels on them. Consider this application where there are 12 buttons arranged again in a grid as shown here, such that all go to the same event handler →

Now when a button is pressed, we cannot look at the text on the button since there is no text ... just different background colors. We will use the **getSource()** method and compare the pressed button with each button in the array. Once we find the button that matches, we will know its row and column and will be able to toggle the background of that button accordingly. Here is the code:



```
import java.awt.*;           // Needed for Color
import java.awt.event.*;    // Needed for ActionListener and ActionEvent
import javax.swing.*;       // Needed for JFrame and JButton

public class ToggleButtonsApp extends JFrame implements ActionListener {
    JButton[][] buttons;    // This stores all buttons

    public ToggleButtonsApp (String title) {
        super(title);
        getContentPane().setLayout(null);

        buttons = new JButton[4][3];
        for(int row=0; row<4; row++) {
            for (int col=0; col<3; col++) {
                buttons[row][col] = new JButton();
                buttons[row][col].setLocation(10+col*55, 10+row*55);
                buttons[row][col].setSize(50,50);
                buttons[row][col].addActionListener(this);

                // Pick a random color for the button
                if (Math.random() < 0.5)
                    buttons[row][col].setBackground(Color.black);
                else
                    buttons[row][col].setBackground(Color.white);
                getContentPane().add(buttons[row][col]);
            }
        }
    }
}
```

```

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(195,275);
    }

    // This is the single event handler for all the buttons
    public void actionPerformed(ActionEvent e) {
        // Find the row and column of the pressed button
        for(int row=0; row<4; row++) {
            for (int col=0; col<3; col++) {
                if (e.getSource() == buttons[row][col]) {
                    System.out.println("You pressed the button at row: " +
                                        row + ", column: " + col + ".");
                    // Now toggle the button's color
                    if (buttons[row][col].getBackground() == Color.black)
                        buttons[row][col].setBackground(Color.white);
                    else
                        buttons[row][col].setBackground(Color.black);
                }
            }
        }
    }

    public static void main(String args[]) {
        ToggleButtonsApp frame = new ToggleButtonsApp("Toggle");
        frame.setVisible(true);
    }
}

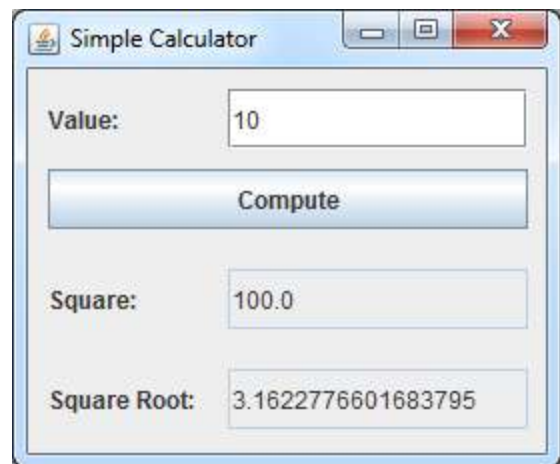
```

Example:



Now let us consider an example of an application that makes use of **JTextFields**. We will create a simple application that allows the user to type in a number into one text field, then press a button and

have the "square" of that number appear in another text field and the "square root" of the number appear in yet another text field as shown here →



In this example, we will only be handling one event ... that of the user pressing the **Compute** button. We will need to extract the text data from the **Value** field.

The **getText()** method in the **JTextField** class allows us to get the text (as a **String** object) that lies in the field. We will need to convert this **String** into a number, such as a **float**, in order to perform computations with it.

In JAVA, we can extract a **float** from a **String** by using the following strategy:

```
float x = Float.parseFloat(aString);
```

This code will convert the **String** (called **aString** in the example) into a **float** (called **x** in the example). There are actually similar ways to convert to other types. Here are some more:

```
int    x = Integer.parseInt(aString);
double x = Double.parseDouble(aString);
boolean x = Boolean.parseBoolean(aString); // "true" to true
```

Once we have the value as a number, we can compute the square and root and then we simply need to put the results into the other two text fields. We do that using **setText()** where we supply a **String** with the result in it. The simplest way to convert a number into a String is to append the number to an empty String object in JAVA as follows:

```
aTextField.setText("" + x);
```

This will work for any number **x**, regardless of whether it is an **int**, **float**, **double**, etc..

One last point ... we will probably want to disable editing in the last two text fields so that the user cannot type into them, since they are "output only" fields. We use the **setEditable(false)** method to do this. Here is the completed code:

```
import java.awt.event.*; // Needed for ActionListener and ActionEvent
import javax.swing.*; // Needed for JFrame, JButton and JTextField

public class CalculatorApp extends JFrame {
    // Text fields to hold the user data and the computed data
    JTextField valueField, squareField, rootField;

    public CalculatorApp(String title) {
        super(title);
        getContentPane().setLayout(null);

        // Add the value label and text field
        JLabel label = new JLabel("Value:");
        label.setLocation(10,10); label.setSize(100, 30);
        getContentPane().add(label);

        valueField = new JTextField();
        valueField.setLocation(100,10); valueField.setSize(150, 30);
        getContentPane().add(valueField);

        // Add the compute button
        JButton computeButton = new JButton("Compute");
        computeButton.setLocation(10,50); computeButton.setSize(240, 30);
        getContentPane().add(computeButton);

        // Add the square label and text field
        label = new JLabel("Square:");
        label.setLocation(10,100); label.setSize(100, 30);
        getContentPane().add(label);

        squareField = new JTextField();
        squareField.setLocation(100,100); squareField.setSize(150, 30);
        squareField.setEditable(false);
    }
}
```

```

getContentPane().add(squareField);

// Add the square root label and text field
label = new JLabel("Square Root:");
label.setLocation(10,150);    label.setSize(100, 30);
getContentPane().add(label);

rootField = new JTextField();
rootField.setLocation(100,150);    rootField.setSize(150, 30);
rootField.setEditable(false);
getContentPane().add(rootField);

// Handle the button click
computeButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (valueField.getText().length() > 0) {
            float value = Float.parseFloat(valueField.getText());
            squareField.setText("" + value * value);
            rootField.setText("" + Math.sqrt(value));
        }
    }
});

setDefaultCloseOperation(EXIT_ON_CLOSE);
setSize(275,230);
}
public static void main(String args[]) {
    CalculatorApp frame = new CalculatorApp("Simple Calculator");
    frame.setVisible(true);
}
}

```

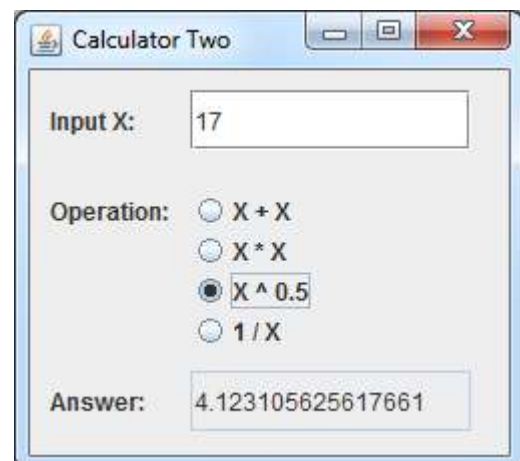
You may have noticed that we did an error-check for the case where no text was in the text field. That is because the `parseFloat()` method generates an ugly error message if the **String** passed in is empty.

Example:



Here is another example of a calculator that can do more operations. It has been set up using a set of **JRadioButtons** which work exactly as **JButtons** do, except that we will add them to a **ButtonGroup** so that only one of the buttons is able to be selected at a time... just like an old-fashioned radio.

To make this work, we will add similar text fields as we did in the previous example. We will also create an array of **JRadioButtons** and have all of them go to the same event handler. Once again, we will search



through the button array to find out which button was pressed and then perform a computation accordingly. In JAVA, a **ButtonGroup** object is used to group buttons together so that only one is on in the group at a time. We simply add **JRadioButtons** to the same button group to get this desired behavior. Here is the code:

```
import java.awt.event.*;    // Needed for ActionListener and ActionEvent
import javax.swing.*;      // Needed for JFrame, JRadioButton and JTextField

public class CalculatorTwoApp extends JFrame implements ActionListener {
    JTextField      valueField, answerField;
    JRadioButton[]  buttons;

    public CalculatorTwoApp(String title) {
        super(title);

        getContentPane().setLayout(null);

        // Add the value label and text field
        JLabel label = new JLabel("Input X:");
        label.setLocation(10,10);    label.setSize(100, 30);
        getContentPane().add(label);

        valueField = new JTextField();
        valueField.setLocation(80,10);    valueField.setSize(140, 30);
        getContentPane().add(valueField);

        // Add the "operation type" radio buttons to the window
        // and to a ButtonGroup so that one is on at a time
        label = new JLabel("Operation:");
        label.setLocation(10,55);    label.setSize(100, 30);
        getContentPane().add(label);

        ButtonGroup operations = new ButtonGroup();
        buttons = new JRadioButton[4];
        String[] buttonLabels = {"X + X", "X * X", "X ^ 0.5", "1 / X"};
        for (int i=0; i<4; i++) {
            buttons[i] = new JRadioButton(buttonLabels[i], false);
            buttons[i].setLocation(80, 60 + i*20);
            buttons[i].setSize(150, 20);
            getContentPane().add(buttons[i]);
            operations.add(buttons[i]);
            buttons[i].addActionListener(this);
        }

        // Add the answer label and text field
        label = new JLabel("Answer:");
        label.setLocation(10,150);    label.setSize(100, 30);
        getContentPane().add(label);

        answerField = new JTextField();
        answerField.setLocation(80,150);
        answerField.setSize(140, 30);
        answerField.setEditable(false);
        getContentPane().add(answerField);
    }
}
```

```

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(255,230);
    }

    // Handle a radio button click
    public void actionPerformed(ActionEvent e) {
        int value = Integer.parseInt(valueField.getText());

        // Find the number of the button that was clicked
        int buttonNumber = 0;
        for (buttonNumber=0; buttonNumber<4; buttonNumber++) {
            if (buttons[buttonNumber] == e.getSource())
                break;
        }

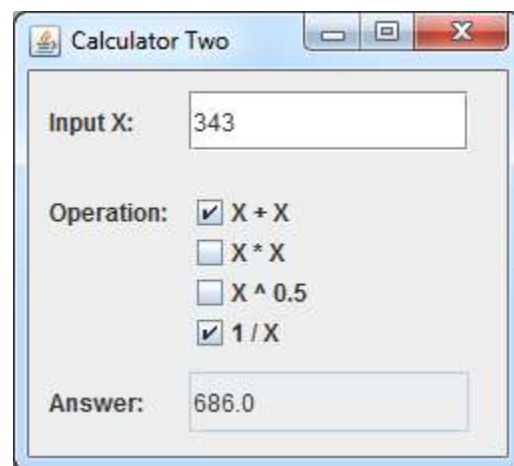
        // Perform the calculation
        double result=0;
        switch (buttonNumber) {
            case 0: result = value + value; break;
            case 1: result = value * value; break;
            case 2: result = Math.sqrt(value); break;
            case 3: result = 1 / (double)value; break;
        }

        // Show the answer
        answerField.setText("" + result);
    }

    public static void main(String args[]) {
        CalculatorTwoApp frame = new CalculatorTwoApp("Calculator Two");
        frame.setVisible(true);
    }
}

```

Note as well that the **JCheckBox** works similar to the **JRadioButton**, except that normally **JRadioButtons** should have only one on at a time, while **JCheckBoxes** may normally have many on at a time. Here is how the window would look if **JCheckBoxes** were used instead (although keep in mind that in this application, it doesn't make sense to have more than one button on at a time). For **JCheckBoxes**, you should NOT add them to a **ButtonGroup**.



Chapter 6

Proper Coding Style Using MVC

What is in This Chapter ?

In this chapter, we will discuss proper coding style. We begin with an explanation of the **Model/View/Controller** paradigm and then show how these can be incorporated together cleanly with a modular coding style. We discuss how to prepare your model classes for use in a GUI. We will then discuss how the **view** and **controller** code of the user interface can be separated cleanly. As a result, this chapter gives a template that you should follow for all of your Graphical User Interface applications.



6.1 Separating Model, View and Controller Components

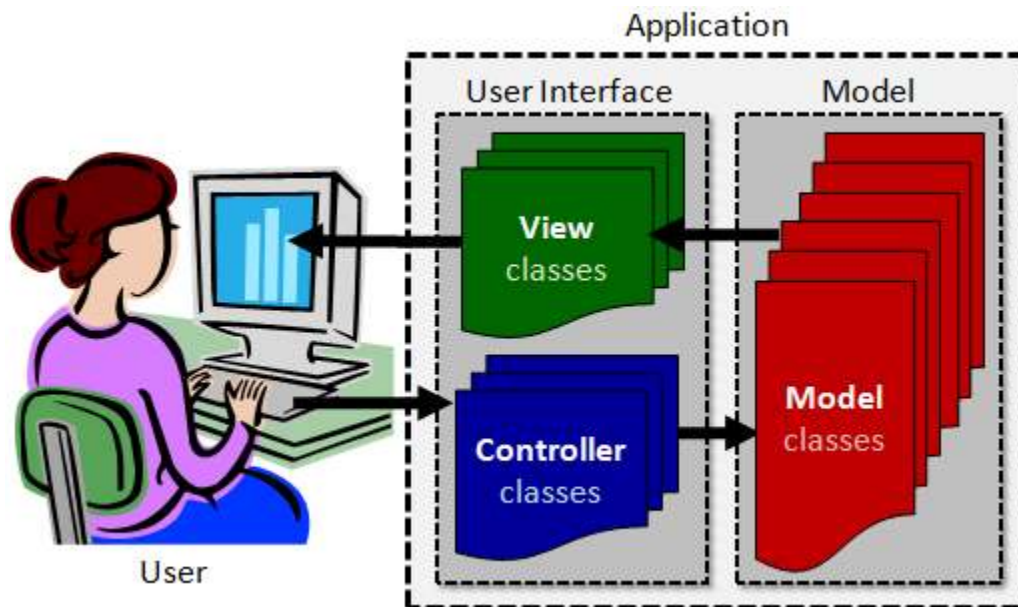
In the previous chapter, we discussed the difference between **model** classes and those classes that are part of the **user interface**. The model classes deal with the *business logic* aspects of the application and the user interface is the "front end" which allows the user to interact with the model classes.

We can further split the user interface classes into two portions called the **view** and the **controller**:

*The **view** displays the necessary information from the model into a form suitable for interaction, typically a user interface element.*

*The **controller** accepts input from the user, modifies the model accordingly.*

The idea is depicted in the picture below. The user sees the **view** of the application and then interacts with the **controller**. Such interaction usually results in the **model** being modified in some way. Then these **model** changes are reflected back to the **view** of the user interface and the user often gets visual feedback that the **model** has changed:



So far, in our examples in the last chapter, we did not have a really useful **model** and the notion of a **view** and a **controller** was not identifiable as all our GUI code was lumped together into one **JFrame** class, with perhaps an extra **JPanel** class.

Now we will discuss the "proper" way of splitting up the **model**, the **view** and the **controller** into a nice & clean modular style that allows us to modify and replace any of the three components cleanly. This arrangement represents what is called the **MVC** software architecture and sometimes referred to as a software **design pattern**. There are 3 main advantages of using the MVC architecture:

1. it decouples the models and views,
2. it reduces the complexity of the overall architectural design and
3. it increases flexibility and maintainability of code.

There are many ways to implement the MVC architecture in your programs. In this course we will consider just one specific way of writing the code. In industry, however, you will see various other ways of implementing the same architecture.

In order to create a well-structured, stable, reliable and maintainable application ... it is necessary to have a properly working model that is designed nicely so that the user interface can connect to it in a simple and safe way. In the next section, we will discuss what is necessary to create this "proper" kind of model.

6.2 Preparing Your Model Classes for the GUI

You already know how to build model classes ... they are the classes that make up your application apart from the user interface components. It is a good idea to prepare your model so that you can interact with it in a simple and clean manner from your user interface. A bank machine, for example, is somewhat pointless unless the underlying **Bank** model is fully operational.

To finalize our model classes, we should decide what kinds of methods should be publically available so that the main application's user interface can access, modify and manipulate the model in meaningful ways.

For example, suppose that we wanted to develop the application that we described earlier that allowed us to make a list of things to purchase at the grocery store. What is the *model* in this application ?

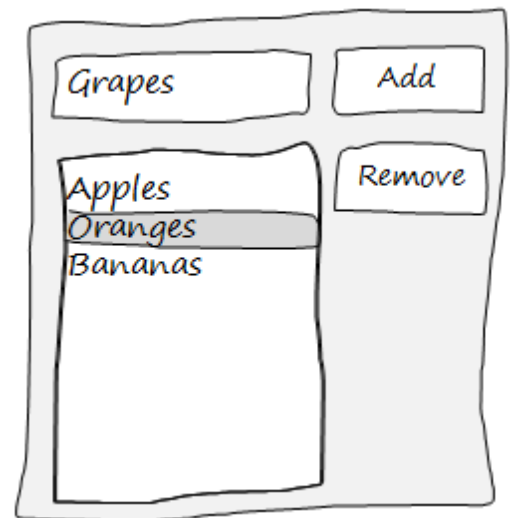
To figure this out, we just have to understand what "lies beneath" the user interface.

What is it that we are displaying and changing ?

It is the list of items.

Let us develop a proper model for this interface.

We can call it **ItemList** and it can keep track of an array of **Strings** that represent the list.



Here is the basic code for this simple model:

```
public class ItemList {
    public final int    MAXIMUM_SIZE = 100;

    private String[]   items;
    private int        size;

    public ItemList() {
        items = new String[MAXIMUM_SIZE];
        size = 0;
    }

    public int getSize() { return size; }
    public String[] getItems() { return items; }
}
```

Looking back at the user interface, it is likely that we will want to add items to the list based on the text that is entered through the text field. The item to be added will likely go at the bottom of the list. So, we should make a public method to do this:

```
public void add(String item) {
    // Make sure that we do not go past the limit
    if (size < MAXIMUM_SIZE)
        items[size++] = item;
}
```

Likewise, the remove button will likely cause the currently selected item in the list to be removed from the list. We will probably remove it according to its index into the list. Here is a public method to do this:

```
public void remove(int index) {
    // Make sure that the given index is valid
    if ((index >= 0) && (index < size)) {
        // Move every item after the deleted one up in the list
        for (int i=index; i<size; i++) {
            items[i] = items[i+1];
        }
        // Reduce the list size by 1
        size--;
    }
}
```

Therefore, here is our completed model:

```
public class ItemList {
    public final int    MAXIMUM_SIZE = 100;

    private String[]   items;
    private int        size;

    public ItemList() {
        items = new String[MAXIMUM_SIZE];
        size = 0;
    }

    public int getSize() { return size; }
    public String[] getItems() { return items; }

    public void add(String item) {
        // Make sure that we do not go past the limit
        if (size < MAXIMUM_SIZE)
            items[size++] = item;
    }

    public void remove(int index) {
        // Make sure that the given index is valid
        if ((index >= 0) && (index < size)) {
            // Move every item after the deleted one up in the list
            for (int i=index; i<size; i++)
                items[i] = items[i+1];
            size--; // Reduce the list size by 1
        }
    }
}
```

Once we have the model implemented with useful methods, it is always a good idea to **test it!** The easiest way to do this is to write a simple test program to try out the various methods. We should write a test program that does a **thorough** testing. We should at least try to add a few items to the list, remove one, remove too many and add too many:

```
public class ItemListTestProgram {
    public static void main(String[] args) {
        // Make a new list
        ItemList groceryList = new ItemList();
        System.out.println("List has " + groceryList.getSize() + " items");

        // Add a few items
        System.out.println("\nAdding Apples, Oranges and Bananas ...");
        groceryList.add("Apples");
        groceryList.add("Oranges");
        groceryList.add("Bananas");
        System.out.println("List has " + groceryList.getSize() + " items");
        System.out.println("Here are the items in the list:");

        for (int i=0; i<groceryList.getSize(); i++)
```

```
        System.out.println(groceryList.getItems()[i]);

// Remove an item
System.out.println("\nRemoving Apples ...");
groceryList.remove(0);
System.out.println("List has " + groceryList.getSize() + " items");
System.out.println("Here are the items in the list:");
for (int i=0; i<groceryList.getSize(); i++)
    System.out.println(groceryList.getItems()[i]);

// Try to remove too many items
System.out.println("\nTrying to remove too many items ...");
groceryList.remove(0);
groceryList.remove(0);
groceryList.remove(0);
groceryList.remove(0);
System.out.println("List has " + groceryList.getSize() + " items");
System.out.println("Here are the items in the list:");
for (int i=0; i<groceryList.getSize(); i++)
    System.out.println(groceryList.getItems()[i]);

// Try to add too many items
System.out.println("\nTrying to add too many items ...");
for (int i=0; i<200; i++)
    groceryList.add("Item# " + i);

System.out.println("List has " + groceryList.getSize() + " items");
System.out.println("Here are the items in the list:");

for (int i=0; i<groceryList.getSize(); i++)
    System.out.println(groceryList.getItems()[i]);
}
}
```

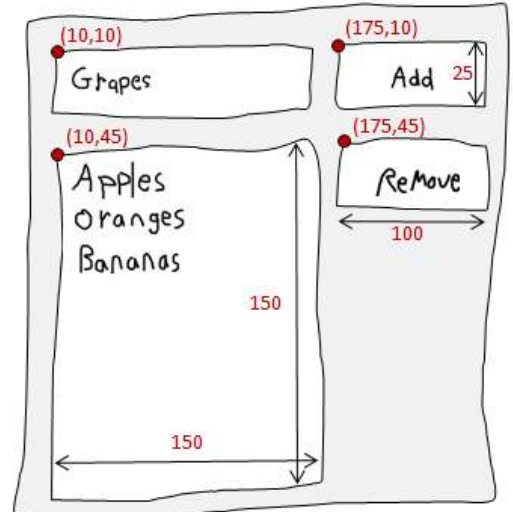
Notice how the test program is nicely formatted with comments indicating what is being tested. Also, notice that there are descriptive print statements that explains what is happening. You should follow a similar style and approach when writing your test programs.

6.3 Developing a *Proper* View

Once you have developed and properly tested the model for your application, you can then begin to write the user interface. Sometimes, however, it may be beneficial to have a rough idea as to what your user interface will do before you develop the model. For example, we did not write the **add()** and **remove()** methods in the **ItemList** class until we realized that we needed them based on what we want our completed application to do.

The next step will be to develop the view for the application. In general, there may be many views in an application, just as there may be many models and controllers. However, we will assume for now that we have a single view.

To keep things simple, we will develop our views as **JPanels** that can be placed onto our **JFrame** windows. Looking back at the chapter on Graphical User Interfaces, you will recall that we made a similar window in our **FruitListApp**. The following code should therefore be easily understood (refer back to chapter 4 if you do not recall):



```
import javax.swing.*;

public class GroceryListView extends JPanel {
    public GroceryListView() {
        // Choose to lay out components manually
        setLayout(null);

        // Add the text field
        JTextField newItemField = new JTextField();
        newItemField.setLocation(10,10);
        newItemField.setSize(150,25);
        add(newItemField);

        // Add the ADD button
        JButton addButton = new JButton("Add");
        addButton.setLocation(175, 10);
        addButton.setSize(100,25);
        add(addButton);

        // Add the REMOVE button
        JButton removeButton = new JButton("Remove");
        removeButton.setLocation(175,45);
        removeButton.setSize(100,25);
        add(removeButton);

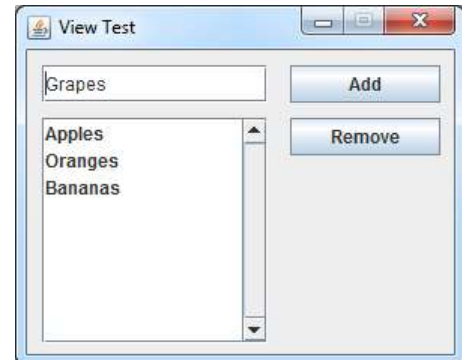
        // Add the JList
        JList aList = new JList();
        JScrollPane scrollPane = new JScrollPane(aList,
            ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
```

```

        ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
scrollPane.setLocation(10,45);
scrollPane.setSize(150,150);
add(scrollPane);

setSize(290, 230); // manually computed sizes
}
}

```



For the purposes of a quick test to make sure that our view is properly formatted, we can create and run a simple program like this:

```

import javax.swing.*;

public class GroceryListViewTestProgram {
    public static void main(String[] args) {
        JFrame frame = new JFrame("View Test");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(290, 230); // manually computed sizes
        frame.setResizable(false);

        frame.getContentPane().add(new GroceryListView());
        frame.setVisible(true);
    }
}

```

Now, for the view to work properly it must refresh its look based on the most up-to-date information in the model. Therefore, we need a way of having the view update itself given a specific model. There are many ways to do this, but a simple way is to write a method called **update()** that will refresh the "look" of the view whenever it is called. Of course, to be able to update, the view must have access to the model. We can pass the model in as a parameter to the view constructor and store it as an attribute of the view:

```

public class GroceryListView extends JPanel {
    private ItemList model; // The model to which this view is attached

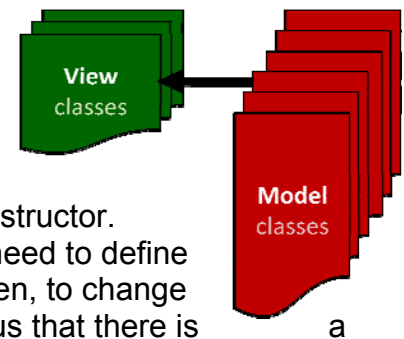
    public GroceryListView(ItemList m) {
        model = m; // Store the model for access later

        ...
    }
}

```

Since the model attribute is simply a reference (i.e., a pointer in memory) to the **ItemList**, then any changes to the **ItemList** will also affect the model stored in this **model** instance variable.

Our **update()** method may be as simple as replacing the entire **JList** with the new items currently stored in the model. So, regardless of what changes take place in the **ItemList** model, we simply read the list of items and re-populate the **JList** with the latest items. To be able to do this, we will need access to that **JList**. However, the **JList** is defined as a local variable in the constructor. In order to be able to access it from the **update()** method, we will need to define the variable outside of the constructor as an instance variable. Then, to change the contents of the **JList**, a quick search in the JAVA API informs us that there is **setListData()** method which will allow us to pass in an array of items to show in the list.



Here is the view code now:

```
public class GroceryListView extends JPanel {
    private ItemList model; // The model to which this view is attached
    private JList aList; // The visible list representing the model

    public GroceryListView(ItemList m) {
        model = m; // Store the model for access later

        ...
        aList = new JList();
        ...
    }
    public void update() {
        aList.setListData(model.getItems());
    }
}
```

Notice how the **JList** is now easily accessible in the **update()** method. One problem, however, is that our model's array is always of size 100 regardless of how many items have been placed in it. The **setListData()** method will end up making a list of 100 items in it ... leaving many blanks. Looking at the API again, there is another **setListData()** method ... but one that takes a **Vector** as a parameter. We will discuss this later in the course. For now, we can add additional code here to make a new array (for display purposes) which has a length exactly equal to the size of the **items** array. Change the **update()** method to this:

```
public void update() {
    // Create and return a new array with the
    // exact size of the number of items in it
    String[] exactList = new String[model.getSize()];
    for (int i=0; i<model.getSize(); i++)
        exactList[i] = model.getItems()[i];

    aList.setListData(exactList);
}
```

Thinking ahead a little, we know that eventually the application will need to respond to user input through pressing the **Add** or **Remove** button, typing in the text field or selecting from the list. This will be part of the controller. However, in order to accomplish this, as you will soon see, we need to allow the controller to access the **JButtons**, the **JList** and the **JTextField**.

We can allow such access by making instance variables for all four window components and then provide **get** methods to access them.

As a final programming aspect of the view class, it is a good idea to call the **update()** method at the end of the constructor. That way, when the view is first created, it can be refreshed right away to show the true state of the model upon startup.

You should follow this same standard approach when designing your views:

```
import javax.swing.*;

public class GroceryListView extends JPanel {
    private ItemList model; // The model to which this view is attached

    // The user interface components needed by the controller
    private JList aList;
    private JButton addButton;
    private JButton removeButton;
    private JTextField newItemField;

    // public methods to allow access to JComponents
    public JList getList() { return aList; }
    public JButton getAddButton() { return addButton; }
    public JButton getRemoveButton() { return removeButton; }
    public JTextField getNewItemField() { return newItemField; }

    public GroceryListView(ItemList m) {
        model = m; // Store the model for access later

        // Choose to lay out components manually
        setLayout(null);

        // Add the text field
        newItemField = new JTextField();
        newItemField.setLocation(10,10);
        newItemField.setSize(150,25);
        add(newItemField);

        // Add the ADD button
        addButton = new JButton("Add");
        addButton.setLocation(175, 10);
        addButton.setSize(100,25);
        add(addButton);

        // Add the REMOVE button
        removeButton = new JButton("Remove");
        removeButton.setLocation(175,45);
        removeButton.setSize(100,25);
```

```

    add(removeButton);

    // Add the JList
    aList = new JList();
    JScrollPane scrollPane = new JScrollPane(aList,
        ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
        ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
    scrollPane.setLocation(10,45);
    scrollPane.setSize(150,150);
    add(scrollPane);

    setSize(290, 230); // manually computed sizes

    // Call update() to make sure model contents are shown
    update();
}

// Update the view to show the model's state
public void update() {
    // Create and return a new array with the
    // exact size of the number of items in it
    String[] exactList = new String[model.getSize()];
    for (int i=0; i<model.getSize(); i++)
        exactList[i] = model.getItems()[i];
    aList.setListData(exactList);
}
}

```

We can run a quick test to make sure that this is working...

```

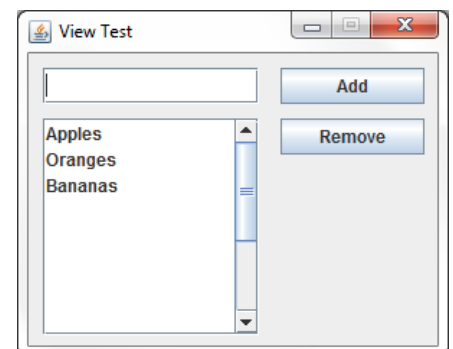
import javax.swing.*.*;

public class GroceryListViewTestProgram2 {
    public static void main(String[] args) {
        ItemList groceryList = new ItemList();
        groceryList.add("Apples");
        groceryList.add("Oranges");
        groceryList.add("Bananas");

        GroceryListView aView = new GroceryListView(groceryList);

        JFrame frame = new JFrame("View Test");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(290, 230); // manually computed sizes
        frame.setResizable(false);
        frame.getContentPane().add(aView);
        frame.setVisible(true);
    }
}

```

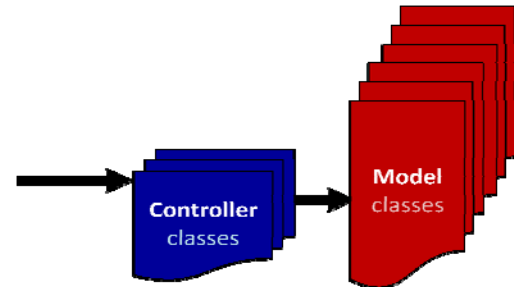


The window should come up now with some fruits listed in the list. We will add some more functionality to the **View** later, but for now, we need to understand how to complete the final portion of our MVC application... the controller ...

6.4 Developing a *Proper* Controller

The final piece of our application is the controller. The controller is responsible for taking in user input and making changes to the model accordingly. These changes can then be refreshed with a simple call to **update()** in the view class.

The controller will be our **JFrame** class, representing the whole application. It will tie together the model and the view. In addition the controller is where we "hook up" our event handlers to handle the user interaction. It will handle all user input and then change the model accordingly ... updating the view afterwards.



To begin, we can define the class such that it creates a new view and a new model. Here is the basic structure ... we will be adding the event handlers one by one. Take note of how cleanly separated the model and the view are ... as they are stored separately as attributes of the controller:

```
import javax.swing.*;           // Needed for JFrame
import java.awt.event.*;       // Need soon for ActionListener
import javax.swing.event.*;    // Need soon for ListSelectionListener, DocumentListener

public class GroceryListApplication extends JFrame {
    private ItemList      model; // The model to which this view is attached
    private GroceryListView view; // The view that shows the state of the model

    public GroceryListApplication(String title) {
        super(title); // Sets the title of the window

        // Create the model and view
        model = new ItemList();
        view = new GroceryListView(model);

        // Add the view
        getContentPane().add(view);

        // Add the event handlers
        // ... coming soon ...

        // Manually computed size
        setSize(290, 230);
        setResizable(false);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    // This is where the program begins
    public static void main(String[] args) {
        JFrame frame = new GroceryListApplication("My Grocery List");
        frame.setVisible(true);
    }
}
```

Now it is time to get everything working. We will approach this slowly by adding functionality as we go along. Make sure you understand where the new pieces of code will "fit" into the three classes that we already have.

Adding Items

To add an item to the Grocery List, we will require the user to type the item into the text field and then press the **Add** button. Let us write an event handler for when the **Add** button is pressed. It will need to get the contents of the text field and then insert that string as a new item in the model. Then the view should be updated. So we should add this method to the controller (i.e., the main application):

```
// The Add Button event handler
private void handleAddButtonPress() {
    model.add(view.getNewItemField().getText());
    view.update();
}
```

Notice that this method is **private**, since no external classes should be calling it. The code does two main things that ALL of your event handlers should do:

1. Change the Model
2. Update the View

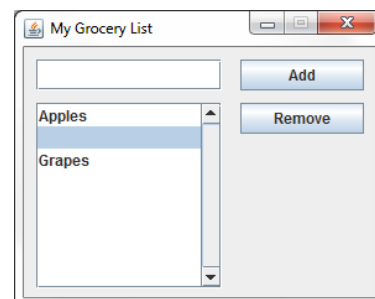


Now, we need to "plug it in" to the **JButton**. We will need to add an **ActionListener** to the button. Insert the following into the controller's constructor:

```
view.getAddButton().addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        handleAddButtonPress();
    }});
```

Notice how the **JButton** is accessed from the view. The listener is added and points to the event handler method that we just wrote. The code should now work and allow new items to be added to the list.

However, a slight problem occurs when the user does not have anything typed into the text field and then presses the **Add** button. A "blank" item is added to the list. This is not pleasant.



We can ensure that no blank items are added by altering our code a little:

```
private void handleAddButtonPress() {
    String text = view.getNewItemField().getText().trim();
    if (text.length() > 0) {
        model.add(text);
        view.update();
    }
}
```

You may have noticed the **trim()** method added here. This is a **String** method that removes any leading and trailing space and tab characters. That will ensure that we do not add any items consisting of only spaces or tab characters.

Removing Items

Similarly, to remove an item from the Grocery List, we will require the user to select the item from the list and then press the **Remove** button. Let us write an event handler for when the **Remove** button is pressed. It will need to get the index of the selected item from the list and call the model's **remove()** method using this index. Then the view should be updated. To get the selected item from the list, we can look in the JAVA API and determine that the **JList** method we need to call is **getSelectedIndex()**. This method returns -1 if nothing is selected, so we should handle that. So we should add this method to the controller (i.e., the main application):

```
// The Remove Button event handler
private void handleRemoveButtonPress() {
    int index = view.getList().getSelectedIndex();
    if (index >= 0) {
        model.remove(index);
        view.update();
    }
}
```

Of course, to get it to work, we need to insert the following into the controller's constructor:

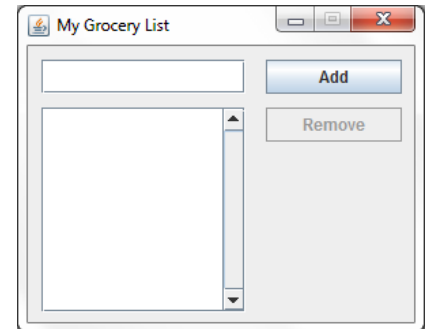
```
view.getRemoveButton().addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        handleRemoveButtonPress();
    }
});
```

We are now done with the basic functionality of our program. We could stop here. However, we will continue with some additional fine-tuning...

Disabling the Remove Button

It would be a good idea to disable the **Remove** button when nothing has been selected in the list. That way the user knows visually that the **Remove** operation is now valid until something is selected. It is a good form of feedback to the user and it makes the user interface more intuitive to use.

Since this is simply a visual change, we could simply add a line to the **update()** method in the view class:



```
public void update() {
    // Create and return a new array with the
    // exact size of the number of items in it
    String[] exactList = new String[model.getSize()];
    for (int i=0; i<model.getSize(); i++)
        exactList[i] = model.getItems()[i];
    aList.setListData(exactList);

    // enable/disable the remove button accordingly
    removeButton.setEnabled(aList.getSelectedIndex() >= 0);
}
```

Notice that if the list has something in it, then the selected index will be 0 or more. We can use this boolean result to set the button to be enabled or disabled. If we start up the application, the **Remove** button will be disabled properly. When should it be re-enabled? According to our **update()** method, whenever something is selected from the list it will be enabled. However, we need to ensure that the **update()** method is called when the user selects something from the list. Whenever the user clicks in the list, we can simply update the view to ensure that the **Remove** button is re-enabled. Here is the simple event handler to put into the controller class:

```
// The List click event handler
private void handleListSelection() {
    view.update();
}
```

There are a few ways to cause this to occur. The simplest is to add a **mousePressed** event handler to the **JList**. Again, we plug it in by adding this to the constructor:

```
view.getList().addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        handleListSelection();
    }});
```

A minor issue is that when we update the list, the currently selected item becomes unselected. We can fix this by remembering what is selected before we update the list and then re-select that item afterwards. In the view, we can alter the **update()** method to do this:

```
// Update the view to show the model's state
public void update() {
    //Remember what was selected
    int selectedItem = aList.getSelectedIndex();

    // Now re-populate the list
    String[] exactList = new String[model.getSize()];
    for (int i=0; i<model.getSize(); i++)
        exactList[i] = model.getItems()[i];
    aList.setListData(exactList);

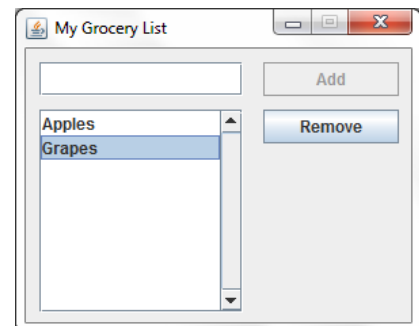
    // Reselecte the selected item
    aList.setSelectedIndex(selectedItem);

    removeButton.setEnabled(aList.getSelectedIndex() >= 0);
}
```

Disabling the Add Button

Finally, it would also be a good idea to disable the **Add** button when nothing is typed in the text field.

To do this, we need to add a single line to the **update()** method in the view that is similar to the one we added to enable/disable the **Remove** button. However, this time we look at the text in the text field:



```
public void update() {
    ...
    addButton.setEnabled(newItemField.getText().trim().length() > 0);
}
```

Of course, once again we need to have the button re-enabled when the user starts typing into the text field. We need an event that occurs when the user types into a text field. Again the event handler is simple:

```
// The text field typing event handler
private void handleTextEntry() {
    view.update();
}
```


Plugging this in to the controller, however, is a little trickier. Modifying the contents of the text field can be done via single character typing or pasting some text or selecting/deleting text etc... Because of the complexity, JAVA decided to create a **Document** object for each **JTextField**. So we will need to implement a **DocumentListener**. Each **JTextField** has a method called **getDocument()** that gets the document that belongs to the text field. We add the listener to that document as follows:

```
view.getNewItemField().getDocument().addDocumentListener(new DocumentListener() {
    public void changedUpdate(DocumentEvent theEvent) { handleTextEntry(); }
    public void insertUpdate(DocumentEvent theEvent) { handleTextEntry(); }
    public void removeUpdate(DocumentEvent theEvent) { handleTextEntry(); }
});
```

Notice that we need to implement three methods to handling inserting, removing and changing of the text in any way. In our situation, we do not care to distinguish between these three since any changes in the text field should generate an **update()**.

Clearing the Text Field

One final alteration to the program would be to clear the text field after an item has been added. Otherwise, after each item has been added, the user will have to delete the text before adding the next item. This can be tedious.

To accomplish this, we simply add one more line to the **Add** button event handler to clear the text:

```
// The Add Button event handler
private void handleAddButtonPress() {
    String text = view.getNewItemField().getText().trim();
    if (text.length() > 0) {
        view.getNewItemField().setText("");
        model.add(text);
        view.update();
    }
}
```

Finally, our application works as desired. We are done.

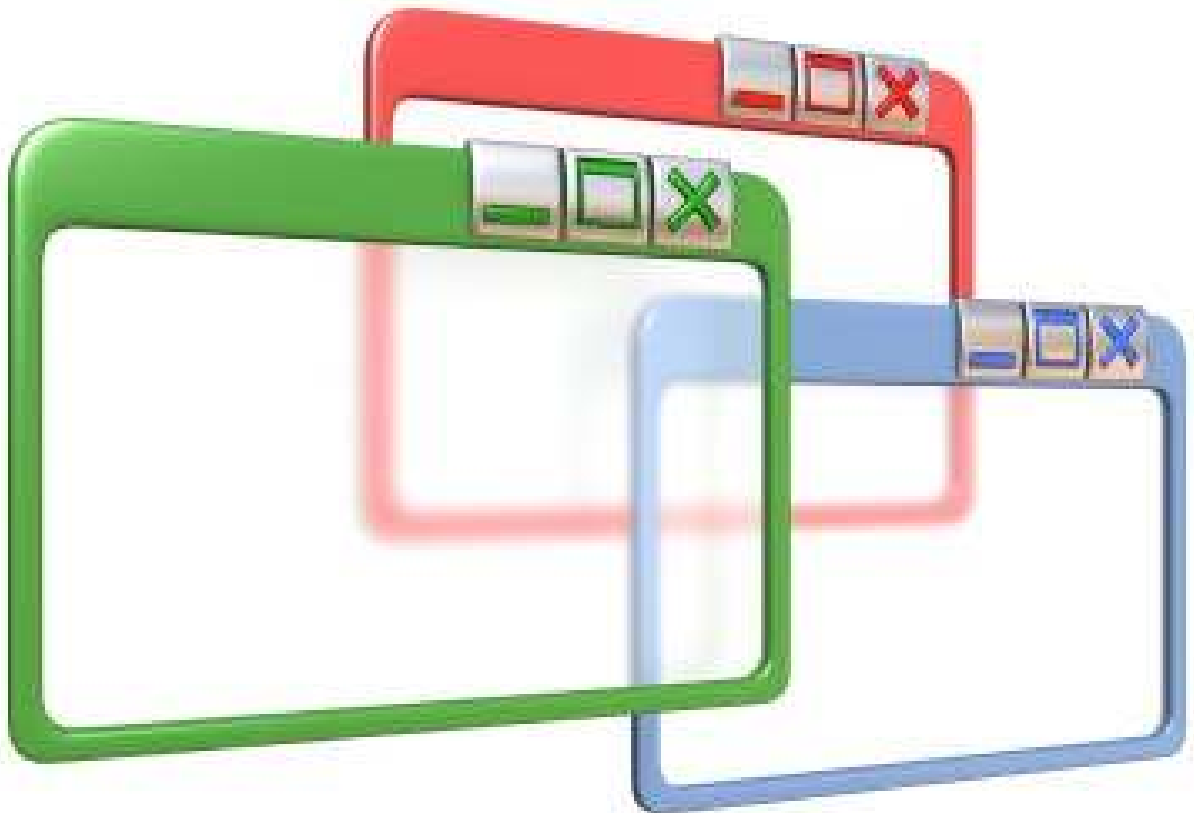
This page was intentionally left blank.

Chapter 7

User Interface Extensions

What is in This Chapter ?

This chapter discusses additional features that can be used to improve and extend your Graphical User Interfaces. It discusses the notion of Layout Managers in java which allow **automatic resizing** of components on the window. The chapter also shows how to add **menus** to your user interfaces as well as develop your own **dialog boxes**.



7.1 Automatic Resizing Using Layout Managers

As you may know ... JAVA was developed for the internet and JAVA applications were initially meant to run as applets within an internet browser. Since browsers are often resized, the application's components need to be rearranged so that they ALL fit on the browser window at all times. In fact, JAVA provides a mechanism called a **Layout Manager** that allows the automatic arrangement (i.e., layout) of the components of an application as the window is resized.

Why should we use a Layout Manager ?

- we would not have to compute locations and sizes for our components
- our components will resize automatically when the window is resized
- our interface will appear "nicely" on all platforms

In JAVA, each layout manager defines methods necessary for a class to be able to arrange **Components** within a **Container**. There are 6 commonly used layout manager classes that implement the **LayoutManager** interface:

FlowLayout, **BoxLayout**, **BorderLayout**, **CardLayout**, **GridLayout**, and **GridBagLayout**

Layouts are set for a panel using the **setLayout()** method. If set to **null**, then no layout manager is used. This is what we have been doing up until this point.

Let us now look at each of these layout managers in turn.

Example (*FlowLayout*):



The simplest layout manager is the **FlowLayout**. It is commonly used to arrange just a few components on a panel. With this manager, components on the window (e.g., buttons, text fields, etc..) are arranged horizontally from left to right ... like lines of words in a paragraph written in English. If no space remains on the current line, components flow (or wrap around) to the next "line". The height of each line is the maximum height of any component on that line. By default, components are centered horizontally on each line, but this can be changed.

There are three constructors that can be used to create a **FlowLayout** manager:

```
public FlowLayout();  
public FlowLayout(int align);  
public FlowLayout(int align, int hGap, int vGap);
```

Here, **align** specifies how the components are to be justified horizontally. It may be any one of three constants:

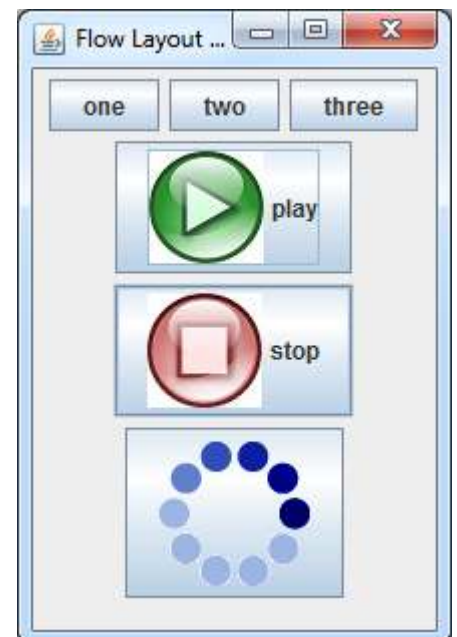
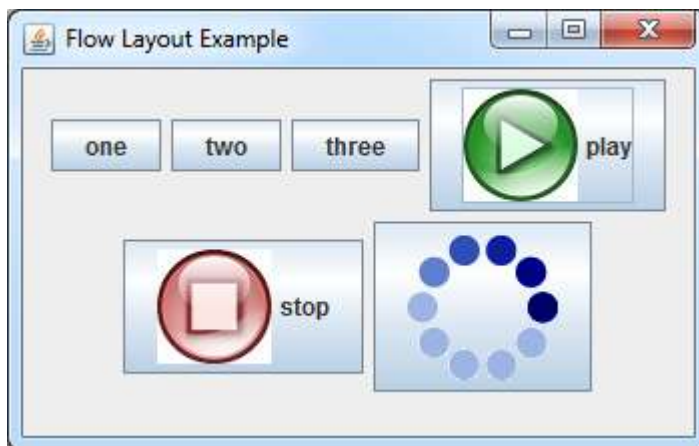
FlowLayout.LEFT, FlowLayout.RIGHT or FlowLayout.CENTER

Also, **hGap** and **vGap** specify the horizontal and vertical margin (in pixels) between components. Here is a simple example that adds 6 buttons (3 with icons) to a panel which uses a **FlowLayout**.

```
import java.awt.*;
import javax.swing.*;

public class FlowLayoutExample extends JFrame {
    public FlowLayoutExample(String title) {
        super(title);
        getContentPane().setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));
        getContentPane().add(new JButton("one"));
        getContentPane().add(new JButton("two"));
        getContentPane().add(new JButton("three"));
        getContentPane().add(new JButton("play", new ImageIcon("GreenButton.jpg")));
        getContentPane().add(new JButton("stop", new ImageIcon("RedButton.jpg")));
        getContentPane().add(new JButton(new ImageIcon("Progress.gif")));
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(500, 200);
    }
    public static void main(String[] args) {
        new FlowLayoutExample("Flow Layout Example").setVisible(true);
    }
}
```

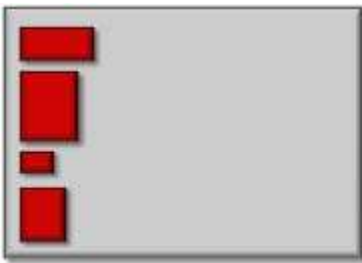
Notice that we can place an image onto a component as an **ImageIcon** object, passing in the name of the **gif** or **jpg** file ... provided that it is in the same directory that this code is running in. Even animated gif files can be used. Here is the result obtained when the application window is resized in different ways ... take notice of how the components wrap around to the next "line":





Keep in mind that the above example just places JButtons as the components, however any components can be used here.

Example (*BoxLayout*):



The **BoxLayout** is also very simple to use. It is similar to the **FlowLayout** in that it arranges components one after another. However, it does not have a wrap around effect. Instead, any components that do not fit on the line are simply not shown. Also, a **BoxLayout** allows you to arrange the components horizontally or vertically.

There is one constructor that can be used to create a **BoxLayout** manager:

```
public BoxLayout(Container panel, int axis);
```

Here, **axis** specifies how the components are to be justified ... either horizontally by using **BoxLayout.X_AXIS** or vertically by using **BoxLayout.Y_AXIS**. The **panel** parameter is the panel that contains the components (i.e., the panel on which we are applying this layout manager). Here is a similar example to that from the **FlowLayout** example:

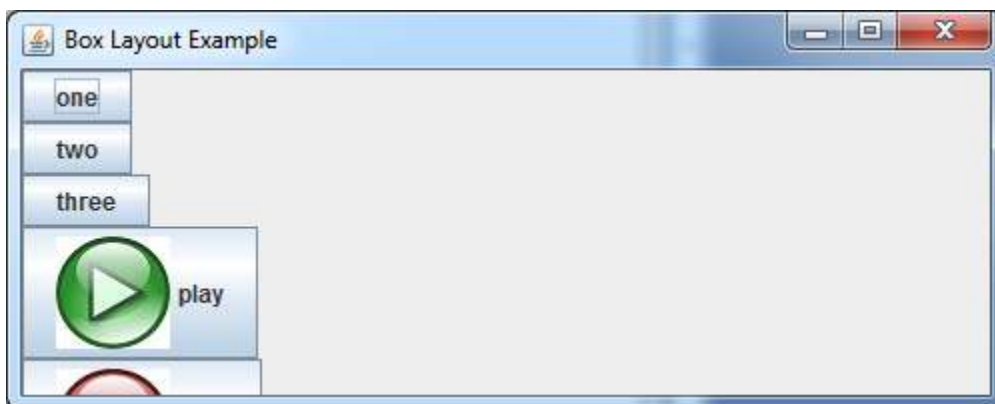
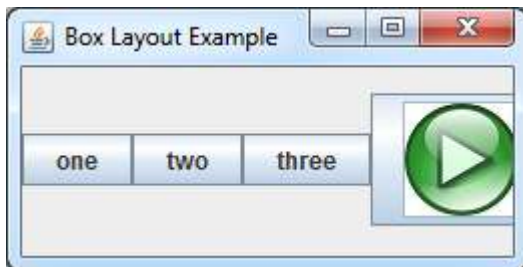
```
import java.awt.*;
import javax.swing.*;

public class BoxLayoutExample extends JFrame {
    public BoxLayoutExample(String title) {
        super(title);
        getContentPane().setLayout(new BoxLayout(this.getContentPane(),
            BoxLayout.X_AXIS));

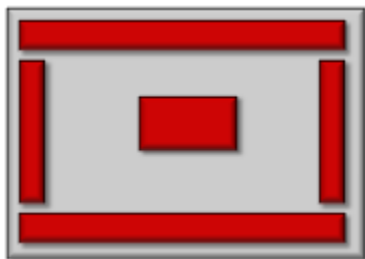
        getContentPane().add(new JButton("one"));
        getContentPane().add(new JButton("two"));
        getContentPane().add(new JButton("three"));
        getContentPane().add(new JButton("play", new ImageIcon("GreenButton.jpg")));
        getContentPane().add(new JButton("stop", new ImageIcon("RedButton.jpg")));
        getContentPane().add(new JButton(new ImageIcon("Progress.gif")));
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(500, 200);
    }

    public static void main(String[] args) {
        new BoxLayoutExample("Box Layout Example").setVisible(true);
    }
}
```

Here is the result obtained when the application window is resized in different ways ... take notice of how the components DO NOT wrap around to the next "line". The first two snapshots below represent an X_AXIS layout while the bottom and right one represent a Y_AXIS layout:



Example (*BorderLayout*):



The **BorderLayout** is a useful layout manager. Instead of re-arranging components, it allows you to place components at one of 5 anchored positions on the window (i.e., north, south, east, west or center). As the window resizes, components stay "anchored" to the side of the window or to its center. The components will grow accordingly. You may place at most one component in each of the 5 anchored positions ... but this one component may be a container such as a **JPanel** that contains other components inside of it. Typically, you do NOT place a component in each of the 5 areas, but choose just a few of the areas.

There are two constructors that can be used to create a **BorderLayout** manager:

```
public BorderLayout()
public BorderLayout(int hgap, int vgap);
```

As with the **FlowLayout**, the **hGap** and **vGap** specify the horizontal and vertical margin (in pixels) between components. When adding components, the **add()** method requires a 2nd parameter indicating the area to add to which must be one of **BorderLayout.NORTH**, **BorderLayout.SOUTH**, **BorderLayout.EAST**, **BorderLayout.WEST** or **BorderLayout.CENTER**. Here is a simple example that adds a **JTextField** to the **CENTER** and a **JPanel** with buttons to the **SOUTH**.

```
import java.awt.*;
import javax.swing.*;

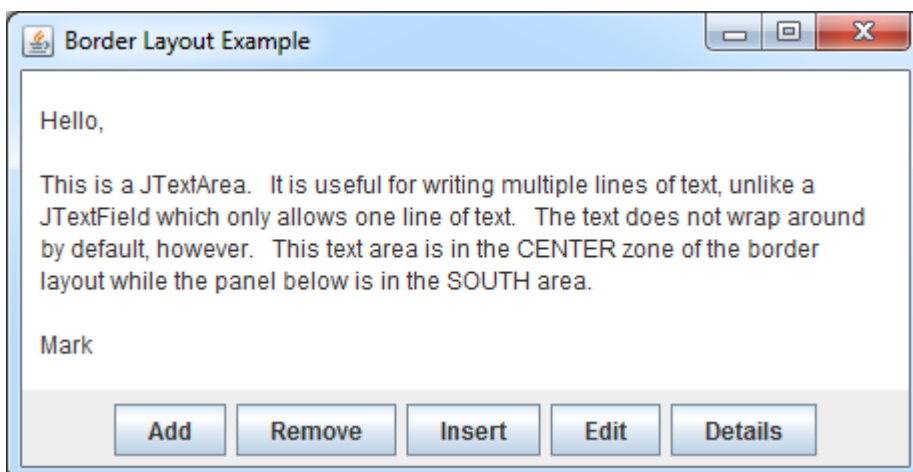
public class BorderLayoutExample extends JFrame {
    public BorderLayoutExample(String title) {
        super(title);

        getContentPane().setLayout(new BorderLayout(2,2));

        JPanel buttonPanel = new JPanel();
        buttonPanel.add(new JButton("Add"));
        buttonPanel.add(new JButton("Remove"));
        buttonPanel.add(new JButton("Insert"));
        buttonPanel.add(new JButton("Edit"));
        buttonPanel.add(new JButton("Details"));
        getContentPane().add(BorderLayout.SOUTH, buttonPanel);
        getContentPane().add(BorderLayout.CENTER, new JTextArea());

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(500, 300);
    }
    public static void main(String[] args) {
        new BorderLayoutExample("Border Layout Example").setVisible(true);
    }
}
```

Here is the result:



We can also make some buttons on the right hand side. Here is an example with a status pane at the bottom as well as a **JPanel** of buttons on the right:

```
import java.awt.*;
import javax.swing.*;

public class BorderLayoutExample2 extends JFrame {
    public BorderLayoutExample2(String title) {
        super(title);

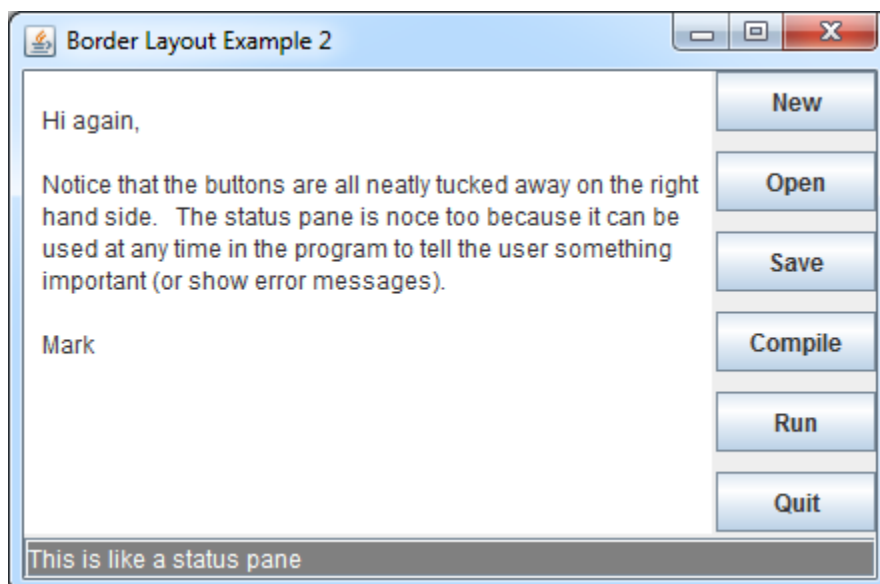
        getContentPane().setLayout(new BorderLayout(2,2));

        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(6,1,10,10));
        buttonPanel.add(new JButton("New"));
        buttonPanel.add(new JButton("Open"));
        buttonPanel.add(new JButton("Save"));
        buttonPanel.add(new JButton("Compile"));
        buttonPanel.add(new JButton("Run"));
        buttonPanel.add(new JButton("Quit"));
        getContentPane().add(BorderLayout.EAST, buttonPanel);

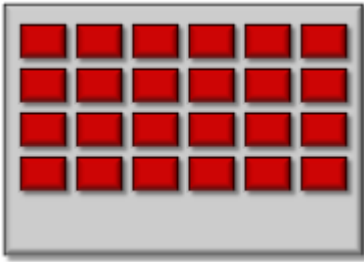
        JTextField statusPane = new JTextField("This is like a status pane");
        statusPane.setBackground(Color.gray);
        statusPane.setForeground(Color.white);
        getContentPane().add(BorderLayout.SOUTH, statusPane);
        getContentPane().add(BorderLayout.CENTER, new JTextArea());

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(500, 300);
    }
    public static void main(String[] args) {
        new BorderLayoutExample2("Border Layout Example 2").setVisible(true);
    }
}
```

Here is the output as the window is resized ... the resizing behavior may not be nice:



Example (GridLayout):



A GridLayout is excellent for arranging a 2-dimensional grid of components (such as buttons on a keypad). It automatically aligns the components neatly into rows and columns. Typically, the components are all of the same size, however you can add different sized components as well. Components are added in sequence one after another until the grid has been filled.

There are two constructors that can be used to create a **GridLayout** manager:

```
public GridLayout(int rows, int columns)
public GridLayout(int rows, int columns, int hGap, int vGap)
```

The **rows** and **columns** parameters tell JAVA how many rows and columns to use for the grid. Again, the **hGap** and **vGap** specify the horizontal and vertical margin (in pixels) between components. When adding components, the **add()** method does not allow you to specify which row and column the component will reside in. Instead, JAVA forces you to add the components one by one and it automatically places them in the grid starting at the top left and moving horizontally until a row is completed ... then moving down to the next row. Here is a simple example that adds some buttons with random background colors of white or black:

```
import java.awt.*;
import javax.swing.*;
public class GridLayoutExample extends JFrame {

    public GridLayoutExample(String title) {
        super(title);

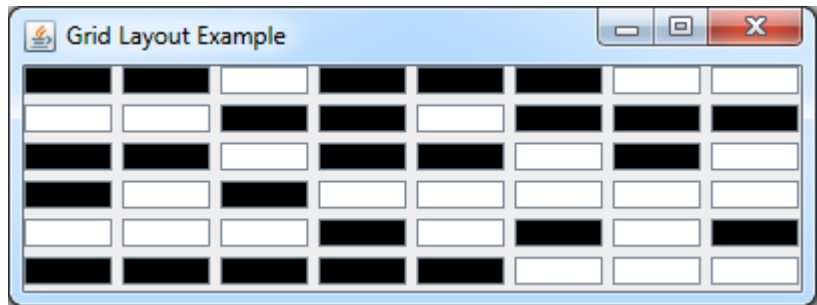
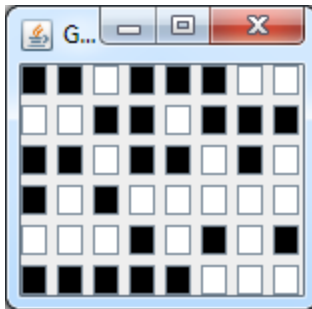
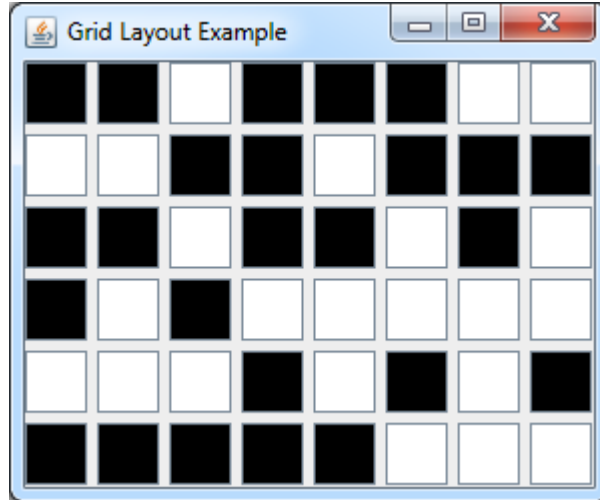
        getContentPane().setLayout(new GridLayout(6,8,5,5));

        for (int row=1; row<=6; row++)
            for (int col=1; col<=8; col++) {
                JButton b = new JButton();
                if (Math.random() < 0.5)
                    b.setBackground(Color.black);
                else
                    b.setBackground(Color.white);
                getContentPane().add(b);
            }

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300, 250);
    }

    public static void main(String[] args) {
        new GridLayoutExample("Grid Layout Example").setVisible(true);
    }
}
```

Here is the result showing a few different resizings:



Example (CardLayout):



The **CardLayout** manager is a little different from the others. It is used to simulate a kind of slideshow because it allows you to have many components in the container but to show only one at a time. A typical application would be to place ImageIcons on a set of labels and have the user click buttons to cycle through the images like a slideshow. In addition to image swapping, you can

actually have entire panels swap in and out to completely alter the user interface at the "drop of a hat" as the expression goes.



There are two constructors that you can use:

```
public CardLayout ()
public CardLayout (int hgap, int vgap)
```

Once the layout manager has been created, it allows you to jump around to various cards (i.e., slides) like a slide projector. Here are the various methods available (the **owner** container is the panel that this layout manager has been applied to):

```
public void first(Container owner)
public void next(Container owner)
public void previous(Container owner)
public void last(Container owner)
public void show(Container owner, String name)
```

Notice that the last method allows you to jump to a specific card/slide, which is uniquely identified by a name. Therefore, when we add cards/slides to the layout, we will supply a unique name for each one.

Here is an example that cycles through 5 images as the user clicks on forward and reverse buttons:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.plaf.basic.BasicArrowButton;

public class CardLayoutExample extends JFrame {
    JPanel    slides;
    CardLayout layoutManager;

    public CardLayoutExample(String title) {
        super(title);

        getContentPane().setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));

        // Create a JPanel with a CardLayout manager for the slides
        slides = new JPanel();
        slides.setBackground(Color.WHITE);
        slides.setBorder(BorderFactory.createLineBorder(Color.BLACK));
        slides.setLayout(layoutManager = new CardLayout(0,0));
        slides.add("1", new JLabel(new ImageIcon("smallDog.jpg")));
        slides.add("2", new JLabel(new ImageIcon("dog1.jpeg")));
        slides.add("3", new JLabel(new ImageIcon("polarBear.jpg")));
        slides.add("4", new JLabel(new ImageIcon("hamsterweights.jpg")));
        slides.add("5", new JLabel(new ImageIcon("catSwim.jpg")));
        getContentPane().add(slides);

        // Now add some slide show buttons for forward and reverse
        JButton rev = new BasicArrowButton(JButton.WEST);
        getContentPane().add(rev);
        JButton fwd = new BasicArrowButton(JButton.EAST);
        getContentPane().add(fwd);

        // Set up the listeners using anonymous classes
        rev.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                layoutManager.previous(slides);
            }
        });

        fwd.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                layoutManager.next(slides);
            }
        });
    }
}
```

```

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(500,500);
    }

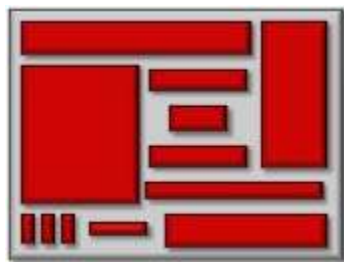
    public static void main(String[] args) {
        new CardLayoutExample("Simple Slide Show").setVisible(true);
    }
}

```

Here is the result showing a few slides in sequence:



Example (GridBagLayout):



The **GridBagLayout** manager is absolutely the most flexible of all the layout managers. It allows you to be very specific in the placement of all components and to indicate exactly how each component is to resize as the window shrinks or grows. However, due to the flexibility of this layout manager, it is MUCH more complicated to use than any of the other layout managers.

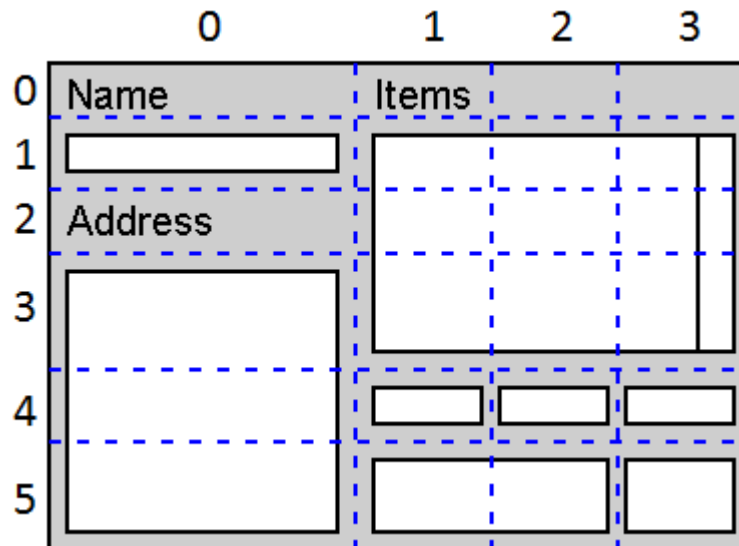
The **GridBagLayout** also arranges components in a grid, but the grid rows and columns are not explicitly defined. Also, the rows and columns may have variable heights and widths. Each component can occupy (i.e., span) multiple rows and columns.

Since there are so many parameters for each component, JAVA supplies a **GridBagConstraints** object that is like a blank form that we must fill-out.

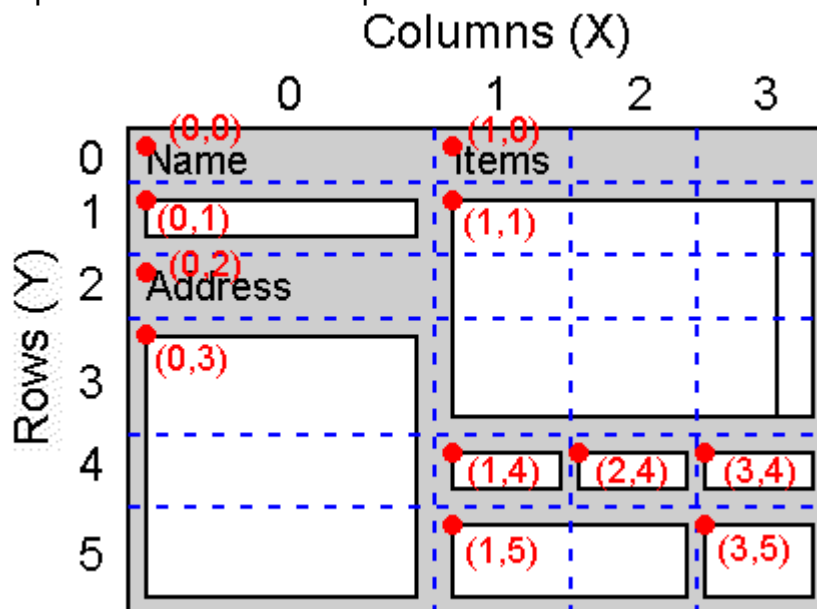
The **GridBagConstraints** objects are used to package together a set of constraints (i.e. parameters) for a particular component. Once the constraints are chosen, they must be set using the **setConstraints()** method for the component. Each constraint has a default which is automatically used if the constraint is not specified.

NAME		INFORMATION
ADDRESS	PHONE	
CITY AND STATE		
MARRIAGE		
DATE OF MARRIAGE		
SIGNATURE		
COMPLETION DATE		COMPLEMENTARY COURSE
SIGNATURE		SIGNATURE

We will now take a look at the many constraints which we can use for each component. The first step when using a **GridBagLayout** is to determine the lines that separate the rows and the columns. We can do this by drawing horizontal and vertical lines whenever there is a change in components in that row or column. Then, we number the rows and columns starting at 0. Here is an example of a window showing the breakdown of the components onto such a grid:



Once we do this, we are ready to specify the parameters for each component. In the above example, there are 11 components (including the text components which are JLabels). For each of these components, we need to determine which grid cell (i.e., row and column numbers) that the top left corner of the component lies in as follows:



This value represents the first parameter that we must set for each component. They are called the **gridx** and **gridy** parameters.

Consider the items box (above) with the topleft corner at (1,1). Here is how we would begin to set the constraints for that single component. Notice that we set the layout manager for the

window's panel and then we start filling in the **constraints** object parameters **gridx** and **gridy**. Afterwards, we apply this to the component by using **setConstraints()**.

```
public class GridBagLayoutExample extends JFrame {
    public CardLayoutExample(String title) {
        super(title);

        GridBagLayout layout = new GridBagLayout();
        getContentPane().setLayout(layout);

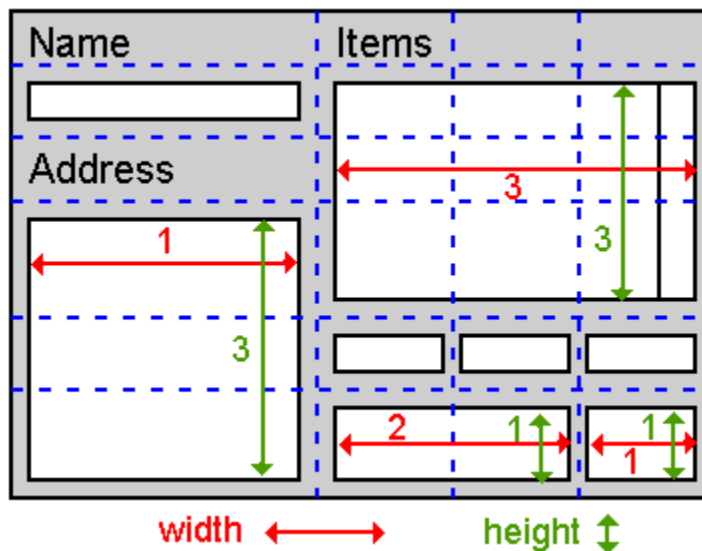
        GridBagConstraints constraints = new GridBagConstraints();

        JList aList = new JList();
        constraints.gridx = 1;
        constraints.gridy = 1;
        ...

        layout.setConstraints(aList, constraints);
        getContentPane().add(aList);

        ...
    }
    ...
}
```

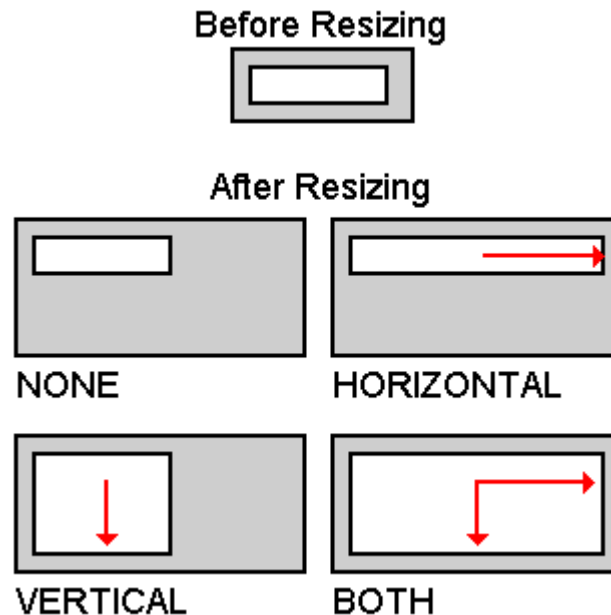
The next step is to determine how many rows and columns each component spans (i.e., takes up). Here are the values for some of the components:



Notice that some of the components take up only 1 row and 1 column (i.e., one grid cell). However, our list takes up space across 3 rows and 3 columns. We set this constraint as the **gridWidth** and **gridHeight** parameters as follows:

```
JList aList = new JList();
constraints.gridx = 1;
constraints.gridy = 1;
constraints.gridwidth = 3;
constraints.gridheight = 3;
...
```

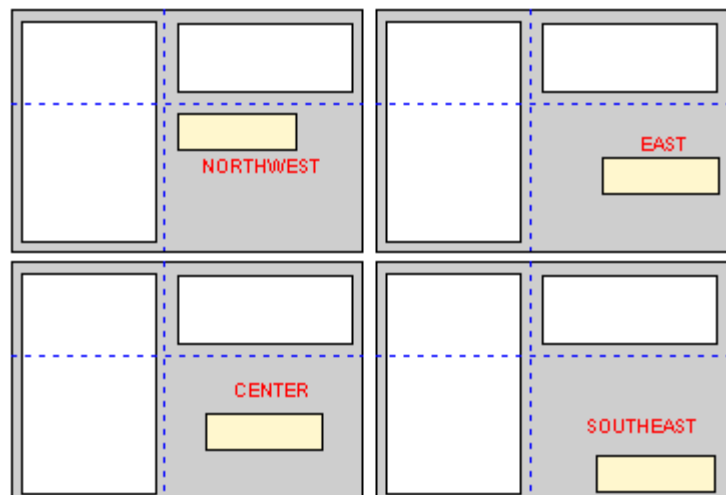
Now ... when the window is enlarged, we need to indicate whether or not the component is to stretch horizontally or vertically to fill up the extra space that becomes available. We can either specify not to fill in the extra space (i.e., `NONE`), to take up only the horizontal (i.e., `HORIZONTAL`) or vertical (i.e., `VERTICAL`) space ... or to take up all space vertically and horizontally (i.e., `BOTH`). Here is what will happen:



We need to set the **fill** parameter to one of these constants. In our list, we perhaps want the list to grow both vertically and horizontally when the window grows, so we can set it to **`GridBagConstraints.BOTH`** as follows:

```
JList aList = new JList();
constraints.gridx = 1;
constraints.gridy = 1;
constraints.gridwidth = 3;
constraints.gridheight = 3;
constraints.fill = GridBagConstraints.BOTH;
...
```

For components that do not have their **fill** set to **`BOTH`**, we need to indicate how the component will move around (or not move) within its cell when the window is resized. We do this by anchoring (i.e., tying) the component to the **`NORTH`**, **`NORTHEAST`**, **`EAST`**, **`SOUTHEAST`**, **`SOUTH`**, **`SOUTHWEST`**, **`WEST`**, **`NORTHWEST`** or **`CENTER`** of the cell using the **anchor** parameter→

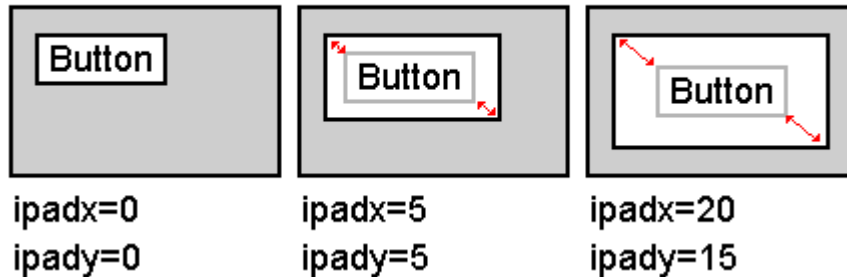


For our list, since we have the fill set to BOTH, it does not matter what anchor we set. However, for something like the top/left label, we would want to anchor it using:

```
constraints.anchor = GridBagConstraints.NORTHWEST;
```

Now, for each component, we can fatten it by supplying a margin around it. There are parameters called **ipadx** and **ipady** that represent the internal padding (i.e., margin in pixels) around the inside of the component:

Setting Minimum Size for Components

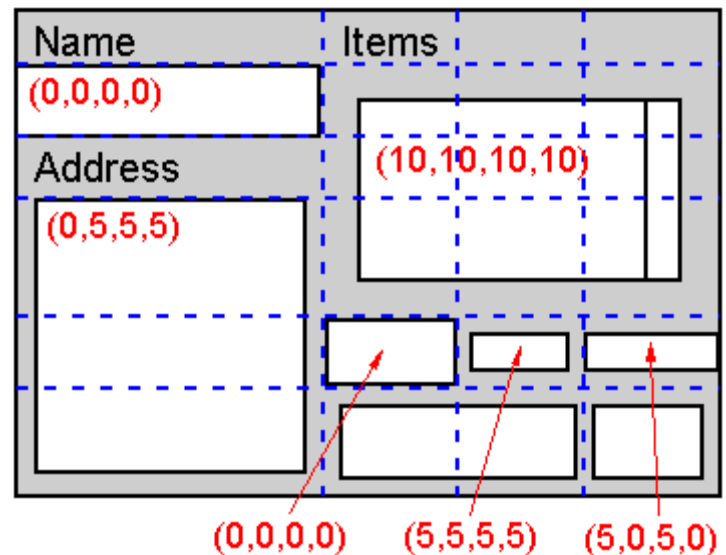


Here is how we can set this for our list:

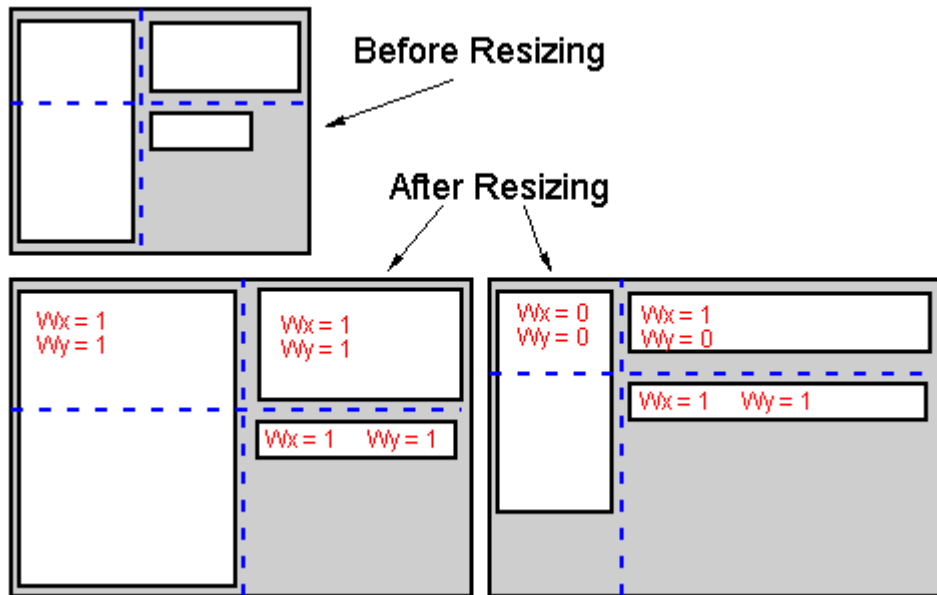
```
JList aList = new JList();
constraints.gridx = 1;
constraints.gridy = 1;
constraints.gridwidth = 3;
constraints.gridheight = 3;
constraints.fill = GridBagConstraints.BOTH;
constraints.ipadx = 5;
constraints.ipady = 5;
...
```

Next, we can specify the margins around the outside of the components. That is, we can specify the margins between components in the same row and column. We do this using the **insets** parameter which is an Insets object which specifies the top, left, bottom and right margins (in pixels) that will offset this component within its cell. Here is how to set it for our list→

```
JList aList = new JList();
constraints.gridx = 1;
constraints.gridy = 1;
constraints.gridwidth = 3;
constraints.gridheight = 3;
constraints.fill = GridBagConstraints.BOTH;
constraints.ipadx = 5;
constraints.ipady = 5;
constraints.insets = new Insets(10, 10, 10, 10);
...
```



Lastly, we can allow components to resize at different rates with respect to one another. To do this, we use the **weightx** and **weighty** parameters which specify the rate at which the component grows with respect to other components in the same row and column.

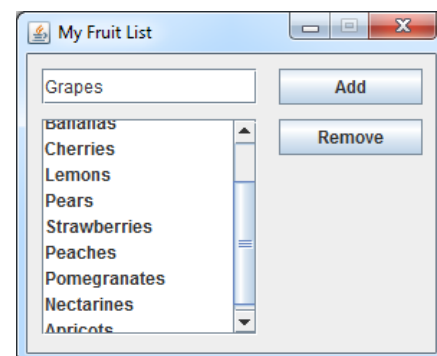


The weight can be sometimes difficult to set as it depends on the fill values of the other components in the same row and column. To begin, it is often best to start all components off with a **weightx** and **weighty** of 1, then adjust them as necessary. It is recommended not to have any row or column where all the weights are set to 0, as this is unpredictable. For our list, perhaps we could set the weights to 10:

```
JList aList = new JList();
constraints.gridx = 1;
constraints.gridy = 1;
constraints.gridwidth = 3;
constraints.gridheight = 3;
constraints.fill = GridBagConstraints.BOTH;
constraints.ipadx = 5;
constraints.ipady = 5;
constraints.insets = new Insets(10, 10, 10, 10);
constraints.weightx = 10;
constraints.weighty = 10;
...
```

The **GridBagLayout** manager can be very complicated. It is best to follow the steps shown above by setting the value for each component individually.

Recall our FruitList example shown here →
How can we use a **GridBagLayout** so that the window is resizable?



Here is the code:

```
import java.awt.*;
import javax.swing.*;

public class GridBagLayoutExample extends JFrame {
    public GridBagLayoutExample(String name) {
        super(name);

        GridBagLayout layout = new GridBagLayout();
        GridBagConstraints constraints = new GridBagConstraints();
        getContentPane().setLayout(layout);

        JTextField newItemField = new JTextField();
        constraints.gridx = 0;
        constraints.gridy = 0;
        constraints.gridwidth = 1;
        constraints.gridheight = 1;
        constraints.fill = GridBagConstraints.BOTH;
        constraints.insets = new Insets(12, 12, 3, 3);
        constraints.weightx = 10;
        constraints.weighty = 0;
        layout.setConstraints(newItemField, constraints);
        getContentPane().add(newItemField);

        JButton addButton = new JButton("Add");
        addButton.setMnemonic('A');
        constraints.gridx = 1;
        constraints.gridy = 0;
        constraints.gridwidth = 1;
        constraints.gridheight = 1;
        constraints.fill = GridBagConstraints.HORIZONTAL;
        constraints.insets = new Insets(12, 3, 3, 12);
        constraints.anchor = GridBagConstraints.NORTHWEST;
        constraints.weightx = 0;
        constraints.weighty = 0;
        layout.setConstraints(addButton, constraints);
        getContentPane().add(addButton);

        String[] stuff = {"Apples", "Oranges", "Grapes", "Pineapples", "Cherries"};
        JList itemsList = new JList(stuff);
        JScrollPane scrollPane = new JScrollPane(itemsList,
            ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
            ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        constraints.gridx = 0;
        constraints.gridy = 1;
        constraints.gridwidth = 1;
        constraints.gridheight = 1;
        constraints.fill = GridBagConstraints.BOTH;
        constraints.insets = new Insets(3, 12, 12, 3);
        constraints.anchor = GridBagConstraints.CENTER;
        constraints.weightx = 10;
        constraints.weighty = 1;
        layout.setConstraints(scrollPane, constraints);
        getContentPane().add(scrollPane);
    }
}
```

```

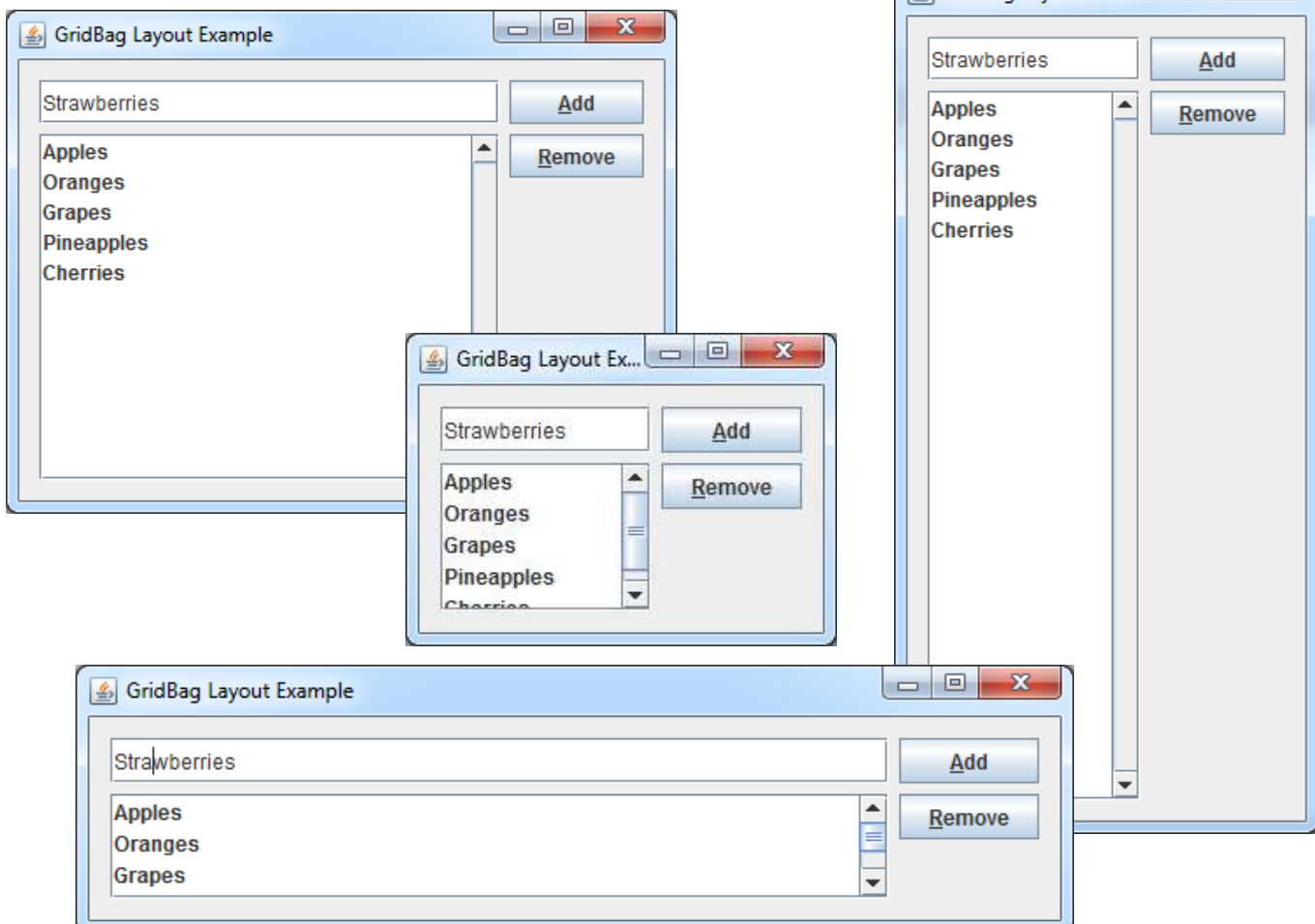
        JButton removeButton = new JButton("Remove");
        removeButton.setMnemonic('R');
        constraints.gridx = 1;
        constraints.gridy = 1;
        constraints.gridwidth = 1;
        constraints.gridheight = 1;
        constraints.fill = GridBagConstraints.HORIZONTAL;
        constraints.insets = new Insets(3, 3, 0, 12);
        constraints.anchor = GridBagConstraints.NORTH;
        constraints.weightx = 0;
        constraints.weighty = 0;
        layout.setConstraints(removeButton, constraints);
        getContentPane().add(removeButton);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(400, 300);
    }

    public static void main(String[] args) {
        new GridBagLayoutExample("GridBag Layout Example").setVisible(true);
    }
}

```

Here is the result as the window is resized in various ways:

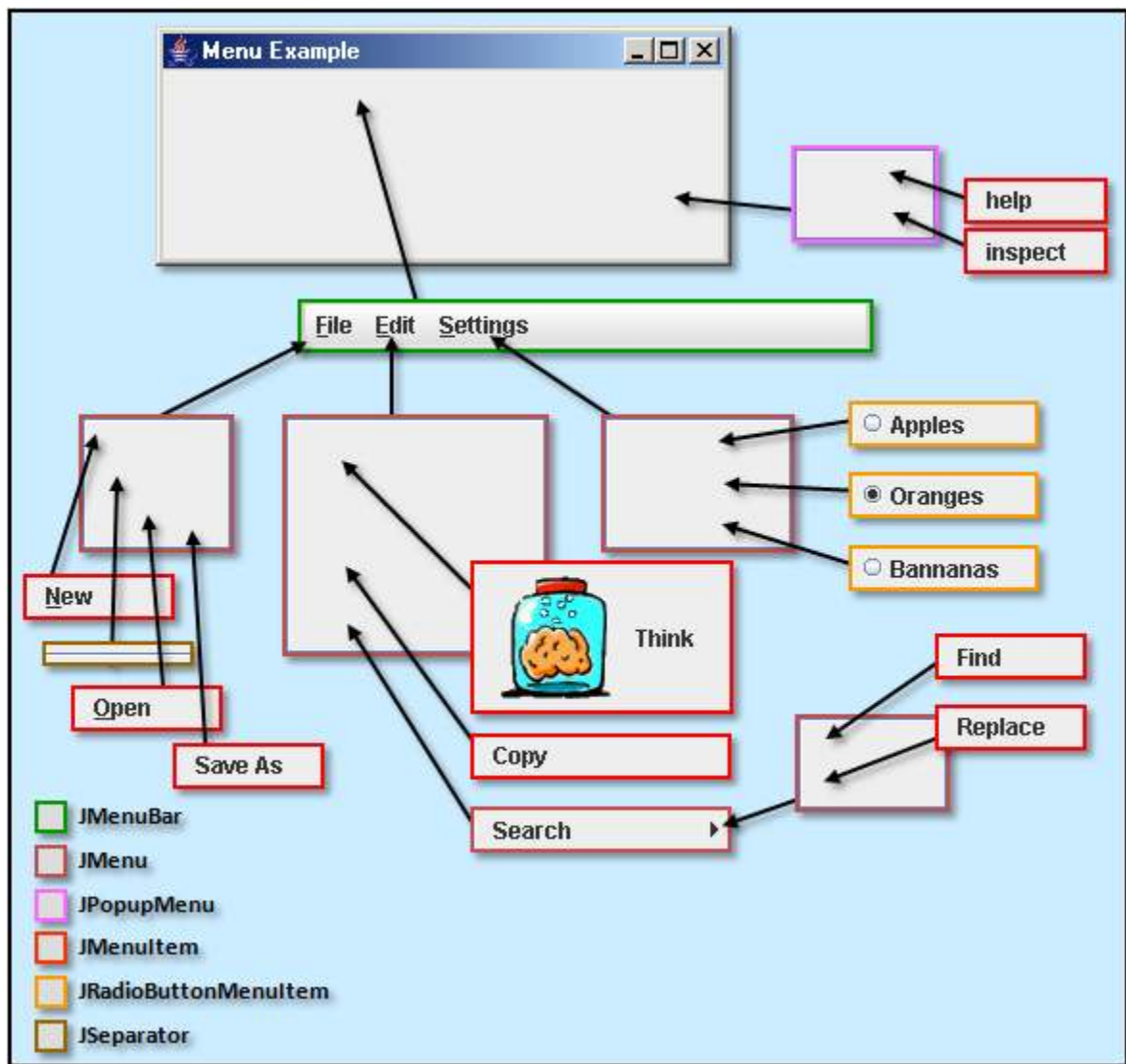


7.2 Adding Menus

A **menu** is a list of commands presented to the user at his/her request. Menus can be attached to a menu bar at the top of an application or they may be pop-up menus that appear anywhere on the screen.



In JAVA, menus are as easy to use as buttons. There are several component classes that may be used including **JMenuBar**, **JMenu**, **JPopupMenu**, **JMenuItem**, **JSeparator** and **JRadioButtonMenuItem**. The diagram below shows how these components are connected together:



Notice that the **JMenuBar** is attached to the main **JFrame** as well as the **JPopupMenu**. The **JMenus** are then added to the **JMenuBar**, or to another menu to form a **cascaded menu** (e.g., the **Search** menu here). The **MenuItems** are simply added to the **JMenus**.

Example:

Consider writing a program to produce the menu hierarchy in the above diagram. We will make a simple **JFrame** with nothing inside it except for the menu bar attached to the top.

A **JMenuBar** is added to a **JFrame** by doing the following in the **JFrame** constructor:

```
JMenuBar myMenuBar = new JMenuBar();
this.setJMenuBar(myMenuBar);
```

Once a menu bar has been created, then **JMenus** can be added to it in a simple manner:

```
JMenu fileMenu = new JMenu("File");
myMenuBar.add(fileMenu);
```

Optionally, we can set the keyboard accelerators (i.e., quick keys) for the menu as well:

```
fileMenu.setMnemonic('F');
```

Once a menu has been created we can add **JMenuItems** and/or **JSeparators** to it. The menus will appear in the order that we add them:

```
JMenuItem newItem = new JMenuItem("New");
JSeparator sepItem = new JSeparator();
fileMenu.add(newItem);
fileMenu.add(sepItem);
```

We can also set the keyboard accelerators for the **JMenuItems** if desired:

```
// This could have been done in the
// constructor: new JMenuItem("New", 'N');
newItem.setMnemonic('N');
```

To get it working, we then add an **ActionListener** to each **JMenuItem**:

```
// they may all go to the same event handler or to separate ones
newItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // Handle the selection of this item from the menu
    }
});
```

We can also add **JRadioButtonMenuItems** to our **JMenus**:

```
JRadioButtonMenuItem rbItem1 = new JRadioButtonMenuItem("Apples");
JRadioButtonMenuItem rbItem2 = new JRadioButtonMenuItem("Oranges");
JRadioButtonMenuItem rbItem3 = new JRadioButtonMenuItem("Bananas");
settingsMenu.add(rbItem1);
settingsMenu.add(rbItem2);
settingsMenu.add(rbItem3);
```

```

rbItem1.addActionListener(this);
rbItem2.addActionListener(this);
rbItem3.addActionListener(this);

// Add them to a button group so that only one is on at a time
ButtonGroup fruits = new ButtonGroup();
fruits.add(rbItem1);
fruits.add(rbItem2);
fruits.add(rbItem3);

```

We can add cascading menus simply by adding a **JMenu** to another **JMenu**:

```

JMenu searchMenu = new JMenu("Search");
JMenuItem findItem = new JMenuItem("Find");
JMenuItem replaceItem = new JMenuItem("Replace");
searchMenu.add(findItem);
searchMenu.add(replaceItem);

```

We then add the cascaded menu to some other **JMenu**:

```

fileMenu.add(searchMenu);

```

Finally, we add a **JPopupMenu** to the **JFrame**:

```

JPopupMenu popupMenu = new JPopupMenu();
JMenuItem helpItem = new JMenuItem("help");
JMenuItem inspectItem = new JMenuItem("inspect");
popupMenu.add(helpItem);
popupMenu.add(inspectItem);

```

To bring up a popup menu, we have to do a bit more work. On a PC, the right mouse button is usually used to bring up a popup menu. This is different on a Mac. JAVA has a method that can determine whether or not the "popup trigger" action (e.g., right mouse click) has just occurred. We can make use of this in our **mouseReleased()** event handler. When we determine that we really do want to bring up the menu, the **show()** method is used ... which lets us specify the component (e.g., panel) on which to pop up the menu along with the x, y position within that component that we want the menu to appear at:

```

myFrame.addMouseListener(new MouseAdapter() {
    public void mouseReleased(MouseEvent e) {
        if (e.isPopupTrigger())
            popupMenu.show(e.getComponent(), e.getX(), e.getY());
    }
});

```

Keep in mind that there are other settings for our **JMenus** and **MenuItems**. To set the **Color** we can do this:

```

anItem.setBackground(Color.red);
anItem.setForeground(Color.yellow);

```

Or to Enable/Disable various items we can do this:

```

anItem.setEnabled(true);
anItem.setEnabled(false);

```

Here is the completed code:

```

import java.awt.event.*;
import javax.swing.*;

public class MenuExample extends JFrame implements ActionListener {

    // Store menu items and popup menu for access from event handlers
    JMenuItem thinkItem, copyItem, newItem, openItem, saveAsItem,
        findItem, replaceItem, appleItem, orangeItem,
        bananaItem, helpItem, inspectItem;

    JPopupMenu popupMenu;

    public MenuExample(String title) {
        super(title);

        // Create the menu bar
        JMenuBar menuBar = new JMenuBar();
        this.setJMenuBar(menuBar); // call on JFrame, not on getContentPane()

        // Create and Add the File menu to the Menu Bar
        JMenu fileMenu = new JMenu("File");
        fileMenu.setMnemonic('F');
        fileMenu.add(newItem = new JMenuItem("New", 'N'));
        fileMenu.add(new JSeparator());
        fileMenu.add(openItem = new JMenuItem("Open", 'O'));
        fileMenu.add(saveAsItem = new JMenuItem("Save As"));
        menuBar.add(fileMenu); // Don't forget to do this
        newItem.addActionListener(this);
        openItem.addActionListener(this);
        saveAsItem.addActionListener(this);

        // Create and Add the Edit menu to the Menu Bar
        JMenu editMenu = new JMenu("Edit");
        editMenu.setMnemonic('E');
        editMenu.add(thinkItem = new JMenuItem("Think", new ImageIcon("brain.gif")));
        editMenu.add(copyItem = new JMenuItem("Copy"));
        menuBar.add(editMenu);
        thinkItem.addActionListener(this);
        copyItem.addActionListener(this);

        // Create and Add the Settings menu to the Menu Bar
        JMenu settingsMenu = new JMenu("Settings");
        settingsMenu.setMnemonic('S');
        settingsMenu.add(appleItem = new JRadioButtonMenuItem("Apples"));
        settingsMenu.add(orangeItem = new JRadioButtonMenuItem("Oranges"));
        settingsMenu.add(bananaItem = new JRadioButtonMenuItem("Bananas"));
        menuBar.add(settingsMenu);

        // Ensure that only one radio button is on at a time
        ButtonGroup fruits = new ButtonGroup();
        fruits.add(appleItem);
        fruits.add(orangeItem);
        fruits.add(bananaItem);
    }
}

```



```

// Create the cascading Search menu on the Settings menu
JMenu searchMenu = new JMenu("Search");
searchMenu.add(findItem = new JMenuItem("Find"));
searchMenu.add(replaceItem = new JMenuItem("Replace"));
editMenu.add(searchMenu);
findItem.addActionListener(this);
replaceItem.addActionListener(this);

// Create and Add items to the popup menu. Notice
// that we do not add the popup menu to anything.
popupMenu = new JPopupMenu();
popupMenu.add(helpItem = new JMenuItem("help"));
popupMenu.add(inspectItem = new JMenuItem("inspect"));
helpItem.addActionListener(this);
inspectItem.addActionListener(this);

// Register the event handler for the popup menu
addMouseListener(new MouseAdapter() {
    public void mouseReleased(MouseEvent e) {
        if (e.isPopupTrigger())
            popupMenu.show(e.getComponent(), e.getX(), e.getY());
    }
});

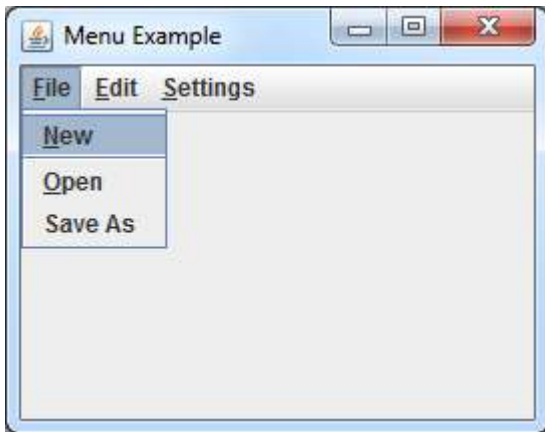
setDefaultCloseOperation(EXIT_ON_CLOSE);
setSize(300, 300);
}

// Handle all menu selections accordingly
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == newItem)
        System.out.println("reacting to NEW selection from menu");
    else if (e.getSource() == openItem)
        System.out.println("reacting to OPEN selection from menu");
    else if (e.getSource() == saveAsItem)
        System.out.println("reacting to SAVE AS selection from menu");
    else if (e.getSource() == copyItem)
        System.out.println("reacting to COPY selection from menu");
    else if (e.getSource() == thinkItem)
        System.out.println("reacting to THINK selection from menu");
    else if (e.getSource() == findItem)
        System.out.println("reacting to FIND selection from menu");
    else if (e.getSource() == replaceItem)
        System.out.println("reacting to REPLACE selection from menu");
    else if (e.getSource() == helpItem)
        System.out.println("reacting to HELP selection from popup menu");
    else if (e.getSource() == inspectItem)
        System.out.println("reacting to INSPECT selection from popup menu");
}

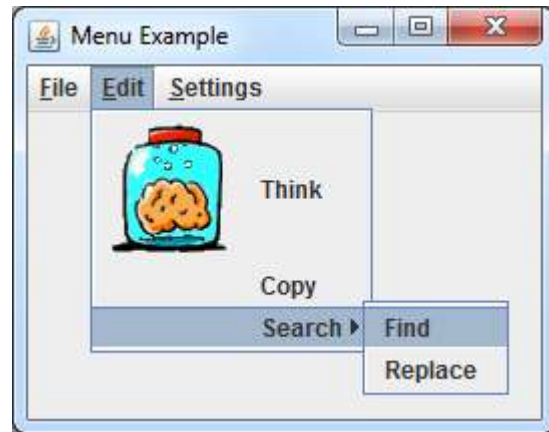
public static void main(String[] args) {
    new MenuExample("Menu Example").setVisible(true);
}
}

```

Here are the resulting screen snapshots showing the various menus:



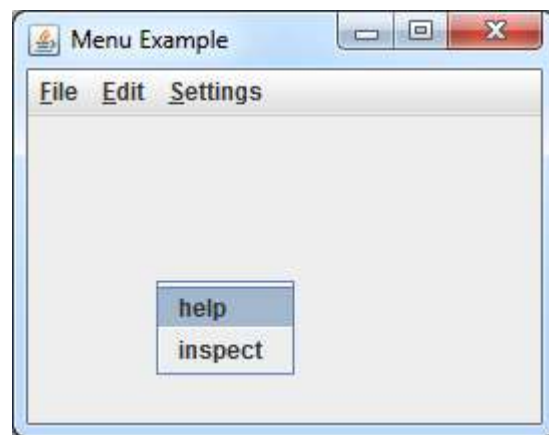
A standard menu



A cascaded menu



A menu with radio buttons



A popup menu

7.3 Standard Dialog Boxes

If a main application window has too many components on it, it will look cluttered and it will not be simple and easy to use. It is a good idea not to display components on your window if they are not needed at that time. For example, a main application may not want to display name, address and phone number fields until the user has selected some action that requires that information to be entered. Usually, this information is placed in a different window that "pops up" when needed.

*A **Dialog Box** is a secondary window (i.e., not the main application window) that is used to interact with the user ... usually to display or obtain additional information.*

So ... a dialog box is another window that can be brought up at any time in your application to interact with the user.

There are various types of commonly used dialog boxes in JAVA:

1. Message Dialog - displays a message indicating information, errors, warnings etc...
2. Confirmation Dialog - asks a question such as yes/no
3. Input Dialog - asks for some kind of input
4. Option Dialog - asks the user to select some option

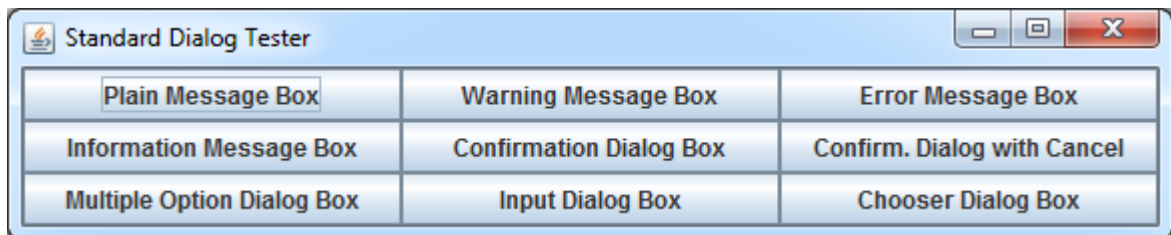
JAVA has a class called **JOptionPane** that can bring up one of these standard dialog boxes. There are many parameters and JAVA allows you to be very flexible in the way that you use them. For instance, there are standard icons that are displayed on these dialog boxes, but you can also make your own.

When using one of these standard dialog boxes, you may specify:

- the frame (owner)
- the title on the dialog box
- the message or question to be asked
- the icon displayed
- the buttons to be shown on the dialog box (i.e. OK, CANCEL, YES, NO)
- a set of options to be asked

Example:

Instead of describing ALL the options and all combinations here, I have decided to just give you a few templates that you can use. Here is some code that tests various standard dialog boxes. It brings up an interface with 9 buttons that allow you to "try out" the boxes. The interface looks as follows:



Here is the code for our test application. Notice the output that appears in the console when running the code. You should be able to figure out how to get information easily from your dialog boxes from this example.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class StandardDialogTestProgram extends JFrame {
    public StandardDialogTestProgram(String title) {
        super(title);
    }
}
```

```

// Make a grid layout for the 9 buttons
getContentPane().setLayout(new GridLayout(3, 3));
JButton aButton;

// Create the button and event handler for the Plain Message Box
getContentPane().add(aButton = new JButton("Plain Message Box"));
aButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null, "This is a plain message !!!",
            "Read This", JOptionPane.PLAIN_MESSAGE);
    }
});

// Create the button and event handler for the Warning Message Box
getContentPane().add(aButton = new JButton("Warning Message Box"));
aButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null, "Don't eat yellow snow.",
            "Warning", JOptionPane.WARNING_MESSAGE);
    }
});

// Create the button and event handler for the Error Message Box
getContentPane().add(aButton = new JButton("Error Message Box"));
aButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null, "Your program stopped working !",
            "Error", JOptionPane.ERROR_MESSAGE);
    }
});

// Create the button and event handler for the Information Message Box
getContentPane().add(aButton = new JButton("Information Message Box"));
aButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null,
            "You better pass the final exam or else ...",
            "Information", JOptionPane.INFORMATION_MESSAGE);
    }
});

// Create the button and event handler for the Confirmation Dialog Box
getContentPane().add(aButton = new JButton("Confirmation Dialog Box"));
aButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        int result = JOptionPane.showConfirmDialog(null,
            "Do you want me to erase your hard drive ?",
            "Answer this Question",
            JOptionPane.YES_NO_OPTION);

        if (result == 0)
            System.out.println("OK, I'm erasing it now ...");
        else
            System.out.println("Fine then, you clean it up!");
    }
});

// Create the button & event handler for Confirmation Dialog Box With Cancel
getContentPane().add(aButton = new JButton("Confirm. Dialog with Cancel"));
aButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        int result = JOptionPane.showConfirmDialog(null,
            "Do you want to overwrite the file ?",

```

```

        "Answer this Question", JOptionPane.YES_NO_CANCEL_OPTION);
    switch(result) {
        case 0: System.out.println("OK, here goes..."); break;
        case 1: System.out.println("Then choose a new name..."); break;
        case 2: System.out.println("I will ask you later..."); break;
    }
    });
}

// Create the button & event handler for Multiple Option Dialog Box
getContentPane().add(aButton = new JButton("Multiple Option Dialog Box"));
aButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        Object[] options = {"Outstanding", "Excellent", "Good", "Fair", "Poor"};
        int result = JOptionPane.showOptionDialog(null,
            "How would you rate your vehicle's performance ?",
            "Pick an Option", JOptionPane.DEFAULT_OPTION,
            JOptionPane.QUESTION_MESSAGE,
            null, options, options[0]);
        if (result > 0) {
            System.out.print("You have rated your vehicle's performance as "
                + options[result]);

            if (result < 3)
                System.out.println("We are glad you are pleased.");
            else
                System.out.println("Please explain why.");
        }
    }
});

// Create the button & event handler for Chooser Dialog Box
getContentPane().add(aButton = new JButton("Chooser Dialog Box"));
aButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        Object[] options = {"Apple", "Orange", "Strawberry", "Banana"};
        Object selectedValue = JOptionPane.showInputDialog(null,
            "Choose your favorite fruit",
            "Fruit Information",
            JOptionPane.INFORMATION_MESSAGE,
            null, options, options[1]);
        System.out.println(selectedValue + "s sure do taste yummy.");
    }
});

// Create the button & event handler for Input Dialog Box
getContentPane().add(aButton = new JButton("Input Dialog Box"));
aButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String in = JOptionPane.showInputDialog("Please input your name");
        System.out.println("Your name is " + in);
    }
});

setDefaultCloseOperation(EXIT_ON_CLOSE);
pack(); //chooses reasonable window size based on component preferred sizes
}

public static void main(String[] args) {
    new StandardDialogTestProgram("Standard Dialog Tester").setVisible(true);
}
}

```

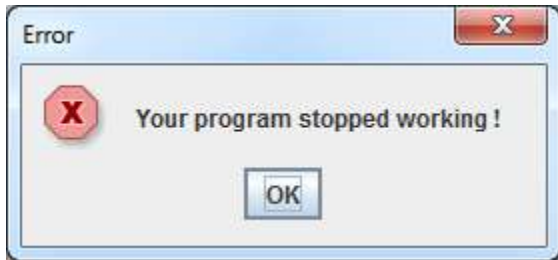
Here are the dialog boxes that will appear.



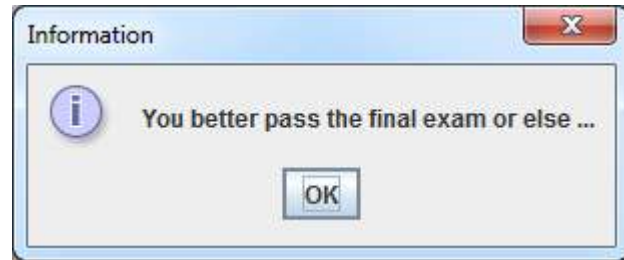
Plain Message Box



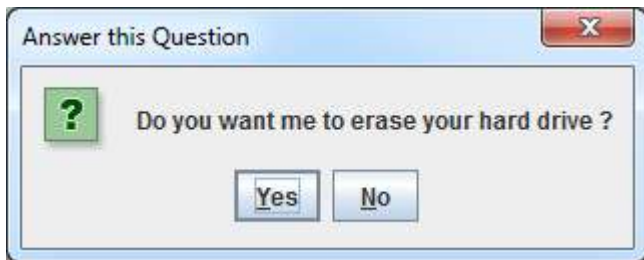
Warning Message Box



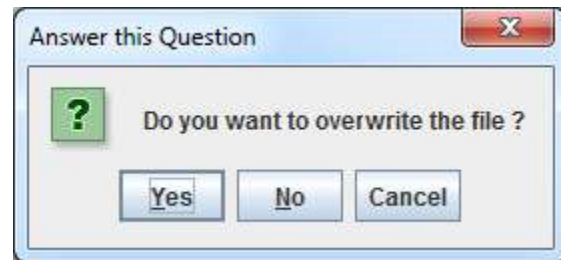
Error Message Box



Information Message Box



Confirmation Dialog Box



Confirmation Dialog Box with Cancel



Multiple Option Dialog Box

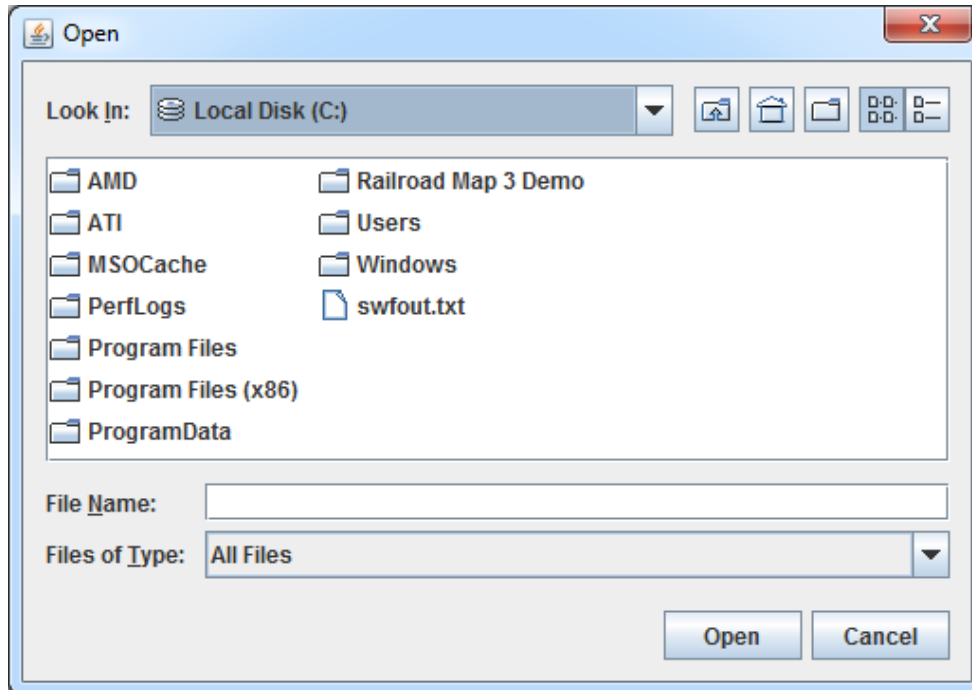


Input Dialog Box



Option Dialog Box

There is another useful **standard** dialog box in JAVA that is used for selecting files. It is called a **JFileChooser**. Here is what it looks like:



Here is some code that opens up a **JFileChooser** box and displays the filename (no path) that the user selects.

```
import javax.swing.*;

public class FileChooserTestProgram {
    public static void main(String[] args) {
        JFileChooser chooser = new JFileChooser();
        int returnVal = chooser.showOpenDialog(null);
        if (returnVal == JFileChooser.APPROVE_OPTION) {
            System.out.println("You chose to open this file: " +
                chooser.getSelectedFile().getName());
        }
    }
}
```

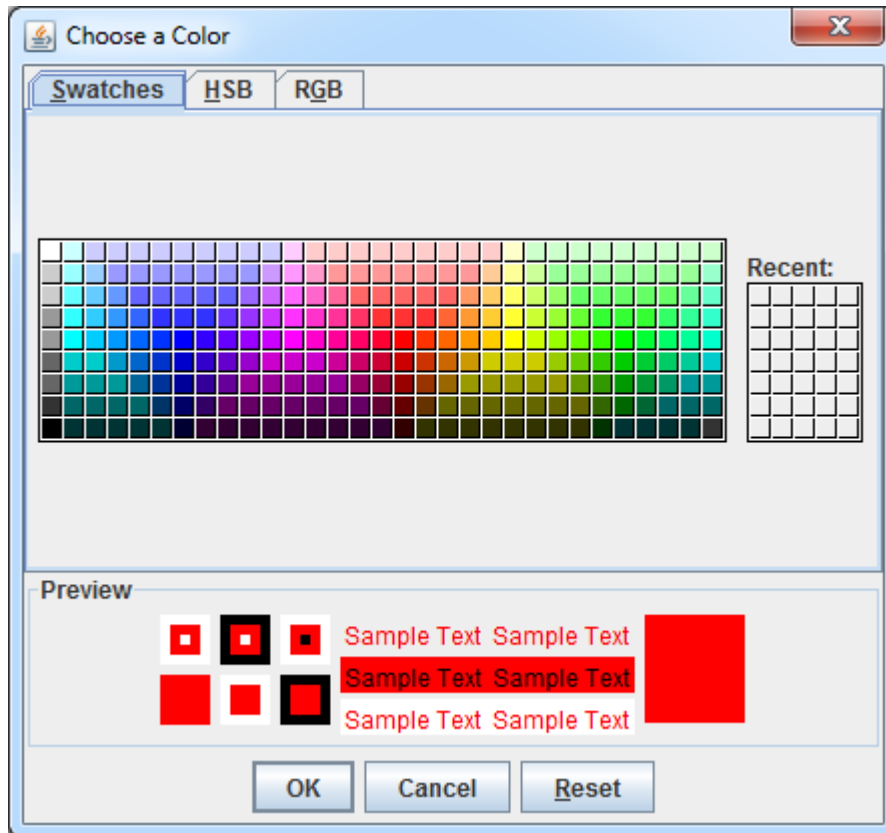
In the above code, the **null** parameter in the **showOpenDialog()** represents the parent window that will bring up this dialog box. Since there are no other windows in this example, it has been set to **null**. There are more options available that allow you to set the filters and starting directories. Take a look at the Swing API. Here, for example, is how to set some filters:

```
FileNameExtensionFilter filter = new FileNameExtensionFilter(
    "JPG & GIF Images", "jpg", "gif");
chooser.setFileFilter(filter);
```

Try adding it to the above code ... you will need to add this to the top of your program:

```
import javax.swing.filechooser.FileNameExtensionFilter;
```

There is also a **JColorChooser** class in JAVA that can be used to bring up a dialog box that allows you to select a color. Here is what it looks like:



You create and add a **JColorChooser** just as you would any other component:

```
import javax.swing.JColorChooser;
import java.awt.Color;

public class ColorChooserTestProgram {
    public static void main(String[] args) {
        Color newColor = JColorChooser.showDialog(
            null, // The parent window
            "Choose a Color", // Title on Dialog Box
            Color.RED); // Initial color selected

        System.out.println("You selected this color: " + newColor);
    }
}
```

Notice that the dialog box returns the color selected when the window is closed.

7.4 Making Your Own Dialog Boxes

There are two modes that a dialog box may be brought up in:

- **modal:** the application window will not respond until this dialog box is closed. This mode forces the user to "deal with" the dialog box information before continuing.
- **non-modal:** the dialog box can remain open while the user works in other windows.

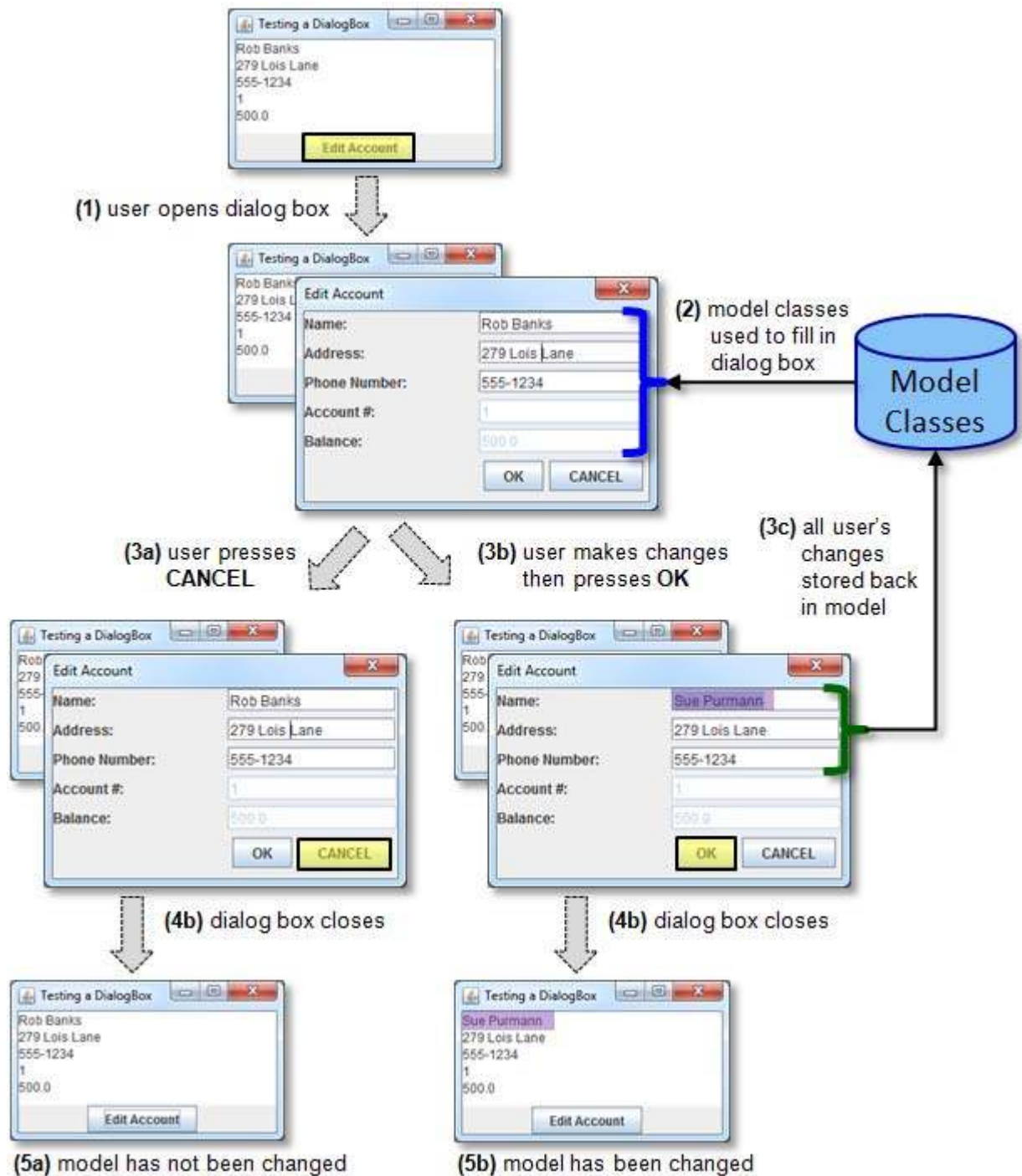
Dialog boxes have an **owner** which is the window that caused it to appear. This allows the dialog box to be closed automatically when the user quits the application from the main window (i.e., all windows belonging to the same application are closed when the application shuts down). Also, when the owner window is minimized, the dialog boxes are also minimized.

There are two important terms pertaining to dialog boxes:

- **Dialog client:** the application that caused the dialog box to appear.
- **Dialog model:** the object(s) that the dialog box should affect.

Normally, an application communicates with its dialog box through a **model** of some kind. That is, the owner opens up a dialog box, passing model-specific information to it. The user may then change this information from the dialog box, which in turn modifies the **model**. When the dialog box is closed, then the main application continues with the modified model objects.

The next page shows a diagram of how everything should work. Notice that the model is used as the "middle-man" between the two windows. That is, when the dialog box is first opened, the model contents are used to populate the components (i.e., fill in the text fields, button selections etc...). The user then makes appropriate changes to the components. When the dialog box is closed with the OK button, the model is updated with these new changes. When the dialog box is closed with the CANCEL button, the model remains unchanged. When either button is clicked, the dialog box closes. The closing of the dialog box using the standard "close" (i.e., X at the top corner) should be treated as a cancel operation.



The dialog box itself is easy to make. It is simply another window. To create your own, simply make it a subclass of **JDialog**:

```
public class MyDialogBox extends JDialog {
    ...
}
```

Then, you can add components and event handlers to this dialog box as if it were a **JPanel**. Typically, you will ensure that there is some combination of ok/apply and cancel/close buttons which are usually located at the bottom right or bottom center of a dialog box.

There are many constructors in the **JDialog** class. We will use the following format for our constructors:

```
public MyDialog(Frame owner, String title, boolean modal, ...) {
    super(owner, title, modal);
    ...

    // Ensure that the dialog box appears close to the main window
    setLocationRelativeTo(owner);
}
```

The **owner** parameter is the client application. In our examples, the **owner** will also need to be a class that implements the **DialogClientInterface** (described soon below). The **title** parameter is what will appear on the dialog box title bar. The **modal** parameter is a **boolean** indicating whether or not the dialog box should be modal. Notice the call to the superclass constructor (this is a standard **JDialog** constructor being called).

We may also want to supply additional model-related parameters to pass information into the dialog box. Often the model itself is passed in as a parameter so that we can (1) fill in the dialog box information based on the current model data, and (2) then we can modify the model as necessary after the user makes changes to the data and presses OK.

In some cases, we may not want the user of the dialog box to decide whether or not it should be modal, nor may we want them to specify the title. We can simply hard-code these into the dialog box if we wish:

```
public MyDialog(Frame owner, ...) {
    super(owner, "Mt Cool Dialog Box", true);
    ...

    // Ensure that the dialog box appears close to the main window
    setLocationRelativeTo(owner);
}
```

In addition to this, we will use the **dispose()** method to dispose of (i.e., close and delete) the dialog box from within our code once we press OK, CANCEL or close the window from the top X.

So dialog boxes are easy to create ... but how do we coordinate the interaction with the main application window and its dialog box ?

The dialog box is defined in a separate class than its owner application. As a result, the **client** (i.e., the application that brought up the dialog box) has no idea what is going on within the dialog box class (nor should it need to know). The **client**, however, usually needs to know whether or not the interaction with the dialog box was **accepted** or whether or not it was **cancelled**. That is, it may need to know whether or not changes were made to the model.

Since the client application does not have access to the internal dialog box classes, the dialog box will need to inform the *client* application as to whether or not the user pressed OK or CANCEL. That means, the dialog box must call one or more methods in the *client* class.

In order to do this cleanly, the dialog box should not need to know who exactly the *client* is. That is, a dialog box in general should allow any application to bring it up and then interact with it. Therefore, we need a general way of communicating with an arbitrary *client* class.

JAVA interfaces are perfect for this situation. We can define the following general interface:

```
public interface DialogClientInterface {
    public void dialogFinished();
    public void dialogCancelled();
}
```

Then, we can have the *client* application implement this interface:

```
public class MyApplication implements DialogClientInterface {
    ...
    public void dialogFinished() {
        ...
    }
    public void dialogCancelled() {
        ...
    }
    ...
}
```

Now, since the *client* application implements the interface, the dialog box has a guarantee that it will have **dialogFinished()** and **dialogCancelled()** methods available to call.

Therefore, in the dialog class, when the OK button is pressed, we can call the *client's* **dialogFinished()** method before closing the dialog box. That will tell the *client* that the model has been changed. Similarly, when CANCEL is pressed (or when the window is closed from the top right X), we can tell the client that the dialog was cancelled. Here is the structure of the dialog box code:

```
public class SomeDialog extends JDialog {
    // A constructor that takes the model and client as parameters
    public SomeDialog(Frame owner, ...){
        ...
    }

    private void okButtonPressed() {
        ...
        ((DialogClientInterface) getOwner()).dialogFinished();
    }
    private void cancelButtonPressed() {
        ...
        ((DialogClientInterface) getOwner()).dialogCancelled();
    }
}
```

Notice how the **owner** (which is a **JFrame**) is type-casted to a **DialogClientInterface** so that the **dialogFinished()** and **dialogCancelled()** methods can be called.

The last piece of information that you will need is that of making the dialog box appear. To have a dialog box appear, you simply create an instance of it and make it visible. Here is some code that would appear in the main application client:

```
public void bringUpDialogBox() {
    MyDialog dialog = new MyDialog (this, "My Dialog", true, model);

    System.out.println("This appears before Dialog box is opened.");

    dialog.setVisible(true);    // Open the dialog box

    System.out.println("This appears after Dialog box is closed...");
    System.out.println("unless the dialog box was non-modal.");
}
```

Example:

Consider having many "buddies" (i.e., friends) that you send e-mails to regularly. You would like to make a nice little electronic address book that you can store the buddy's names along with his/her e-mail addresses. Perhaps you even want to categorize the buddies as being "hot" (i.e., you talk to them often), or not-so-hot.

What exactly is an e-mail buddy? Well we can easily develop a simple model of an **EmailBuddy** as follows:

```
public class EmailBuddy {
    private String name;
    private String address;
    private boolean onHotList;

    // Here are some constructors
    public EmailBuddy() {
        name = "";
        address = "";
        onHotList = false;
    }
    public EmailBuddy(String aName, String anAddress) {
        name = aName;
        address = anAddress;
        onHotList = false;
    }

    // Here are the get methods
    public String getName() { return name; }
    public String getAddress() { return address; }
    public boolean onHotList() { return onHotList; }
```

```

// Here are the set methods
public void setName(String newName) { name = newName; }
public void setAddress(String newAddress) { address = newAddress; }
public void onHotList(boolean onList) { onHotList = onList; }

// The appearance of the buddy
public String toString() {
    return name;
}
}

```

As you may have noticed, there is nothing difficult here ... just your standard "run-of-the-mill" model class. However, this class alone does not represent the whole model for our GUI since we will have many of these **EmailBuddy** objects. So we will need a class to represent the list. We can do this in the same way that we created our grocery item list with an array of **EmailBuddy** objects

```

public class EmailBuddyList {

    public final int    MAXIMUM_SIZE = 100;

    private EmailBuddy[] buddies;
    private int        size;

    public EmailBuddyList() {
        buddies = new EmailBuddy[MAXIMUM_SIZE];
        size = 0;
    }

    // Return the number of buddies in the whole list
    public int getSize() { return size; }

    // Return all the buddies
    public EmailBuddy[] getEmailBuddies() { return buddies; }

    // Get a particular buddy from the list, given the index
    public EmailBuddy getBuddy(int i) { return buddies[i]; }

    // Add an email buddy to the list unless it has reached its capacity
    public void add(EmailBuddy buddy) {
        // Make sure that we do not go past the limit
        if (size < MAXIMUM_SIZE)
            buddies[size++] = buddy;
    }

    // Remove the buddy with the given index from the list
    public void remove(int index) {
        // Make sure that the given index is valid
        if ((index >= 0) && (index < size)) {
            // Move every item after the deleted one up in the list
            for (int i=index; i<size-1; i++)
                buddies[i] = buddies[i+1];
            size--; // Reduce the list size by 1
        }
    }
}

```

```

// Return the number of buddies on the hot list
public int getHotListSize() {
    int count = 0;
    for (int i=0; i<size; i++)
        if (buddies[i].onHotList())
            count++;
    return count;
}

// Get a particular "hot" buddy from the list, given the hot list index
public EmailBuddy getHotListBuddy(int i) {
    int count = 0;
    for (int j=0; j<size; j++) {
        if (buddies[j].onHotList()) {
            if (count == i)
                return buddies[j];
            count++;
        }
    }
    return null;
}
}

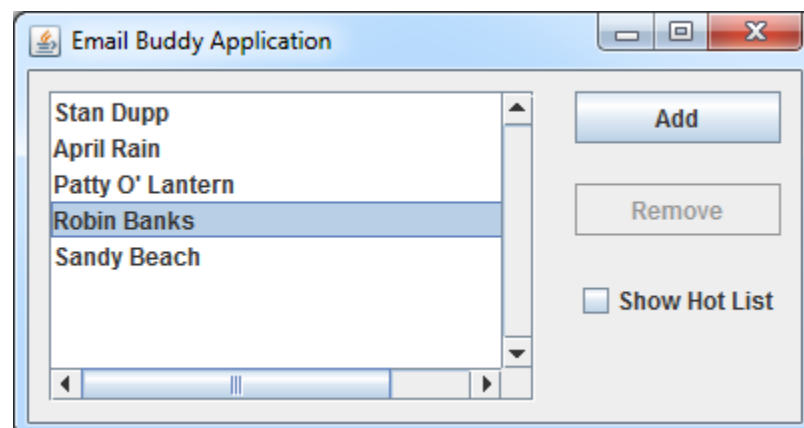
```

Notice that there is a **getSize()** method that is a simple "get" method and there is also a **getHotListSize()** method that returns the number of buddies on the hot list. Notice as well that there are methods to get a buddy at a given index in the array. The method **getHotListBuddy()** will find the i^{th} buddy that is on the hot list. You will see soon why these methods will be useful.

The task now is to design a nice interface for the main application. To start, we must decide what the interface should do. Here is a possible interface:

- A **list** of all buddies is shown (names only)
- We should be able to
 - **Add** and **Remove** buddies from the list
 - **Edit** buddies when their name or email changes
 - Show only those buddies that are "**hot**" or perhaps show all of them

Assume that we have decided upon the following view for the interface:



Notice that the interface does not show the e-mail addresses in the list. It may look cluttered, but we could certainly have done this. Perhaps we could have made a second list box or something that would show the e-mail addresses. Here is a good exercise: make a **JTextField** just beneath the list that will show the e-mail address of the currently selected **EmailBuddy** in the list. This is not hard to do. Nevertheless, it is not necessary for the purposes of explaining this dialog box example.

How can we build the view for this interface ? We will start with a **JPanel**. We will use **GridBagLayout** to allow nice resizing.

```
import java.awt.*;
import javax.swing.*;

public class EmailBuddyPanel extends JPanel {
    private EmailBuddyList    model;        // This is the list of buddies

    private JButton    addButton;
    private JButton    removeButton;
    private JList      buddyList;
    private JCheckBox  hotListButton;

    // These are the get methods that are used to access the components
    public JButton getAddButton() { return addButton; }
    public JButton getRemoveButton() { return removeButton; }
    public JCheckBox getHotListButton() { return hotListButton; }
    public JList getBuddyList() { return buddyList; }

    // This is the default constructor
    public EmailBuddyPanel(EmailBuddyList m){
        super();

        model = m; // Store the model so that the update() method can access it

        // Use a GridBagLayout (lotsa fun)
        GridBagLayout layout = new GridBagLayout();
        GridBagConstraints layoutConstraints = new GridBagConstraints();
        setLayout(layout);

        // Add the buddy list
        buddyList = new JList();
        JScrollPane scrollPane = new JScrollPane(buddyList,
            ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
            ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        layoutConstraints.gridx = 0; layoutConstraints.gridy = 0;
        layoutConstraints.gridwidth = 1; layoutConstraints.gridheight = 3;
        layoutConstraints.fill = GridBagConstraints.BOTH;
        layoutConstraints.insets = new Insets(10, 10, 10, 10);
        layoutConstraints.anchor = GridBagConstraints.NORTHWEST;
        layoutConstraints.weightx = 1.0; layoutConstraints.weighty = 1.0;
        layout.setConstraints(scrollPane, layoutConstraints);
        add(scrollPane);

        // Add the Add button
        addButton = new JButton("Add");
        layoutConstraints.gridx = 1; layoutConstraints.gridy = 0;
        layoutConstraints.gridwidth = 1; layoutConstraints.gridheight = 1;
    }
}
```



```

        layoutConstraints.weightx = 0.0;  layoutConstraints.weighty = 0.0;
        layout.setConstraints(addButton, layoutConstraints);
        add(addButton);

        // Add the Remove button
        removeButton = new JButton("Remove");
        layoutConstraints.gridx = 1;  layoutConstraints.gridy = 1;
        layout.setConstraints(removeButton, layoutConstraints);
        add(removeButton);

        // Add the ShowHotList button
        hotListButton = new JCheckBox("Show Hot List");
        layoutConstraints.gridx = 1;  layoutConstraints.gridy = 2;
        layout.setConstraints(hotListButton, layoutConstraints);
        add(hotListButton);

        // Now update the components by filling them in
        update();
    }
}

```

Of course, we will need to write the **update()** method as well here. Recall that the update method should read from the model and then refresh the "look" of the components. The only components that need their appearance updated is the list and the remove button. The remove button is easily updated as we simply disable it when there is nothing selected in the list:

```
removeButton.setEnabled(buddyList.getSelectedIndex() >= 0);
```

The list is more complicated. First of all, we need to populate the list with the most recent data. Recall that we did something similar in the grocery list example. We need to create an appropriate sized array and then fill it up with email buddies and then set the list data:

```

EmailBuddy[]  exactList;

exactList = new EmailBuddy[model.getSize()];
for (int i=0; i<model.getSize(); i++)
    exactList[i] = model.getBuddy(i);
buddyList.setListData(exactList);

```

However, things are a little more difficult now. If we have the hot list button selected, then we do not want all the buddies ... instead we want only those on the hot list:

```

exactList = new EmailBuddy[model.getHotListSize()];
for (int i=0; i<model.getHotListSize(); i++)
    exactList[i] = model.getHotListBuddy(i);
buddyList.setListData(exactList);

```

We can use an **IF** statement to select the appropriate code:

```

EmailBuddy[]  exactList;
if (hotListButton.isSelected()) {
    exactList = new EmailBuddy[model.getHotListSize()];
}

```

```

        for (int i=0; i<model.getHotListSize(); i++)
            exactList[i] = model.getHotListBuddy(i);
    }
    else {
        exactList = new EmailBuddy[model.getSize()];
        for (int i=0; i<model.getSize(); i++)
            exactList[i] = model.getBuddy(i);
    }
    buddyList.setListData(exactList);

```

We will also need to ensure that we select the selected item each time we make an update. That is, if we were to select an item from the list and then update ... we want to make sure that the item remains selected. At this point, when we refresh the list contents, the selected item does not remain selected. So, we will need to remember which item was selected and then reselect it again after the list is re-populated. Here is the final **update()** method that must be added to the view code:

```

// Update the components so that they reflect the contents of the model
public void update() {
    //Remember what was selected
    int selectedItem = buddyList.getSelectedIndex();

    // Now re-populate the list by creating and returning a new
    // array with the exact size of the number of items in it.
    EmailBuddy[] exactList;
    if (hotListButton.isSelected()) {
        exactList = new EmailBuddy[model.getHotListSize()];
        for (int i=0; i<model.getHotListSize(); i++)
            exactList[i] = model.getHotListBuddy(i);
    }
    else {
        exactList = new EmailBuddy[model.getSize()];
        for (int i=0; i<model.getSize(); i++)
            exactList[i] = model.getBuddy(i);
    }
    buddyList.setListData(exactList);

    // Reselect the selected item
    buddyList.setSelectedIndex(selectedItem);

    // enable/disable the remove button accordingly
    removeButton.setEnabled(buddyList.getSelectedIndex() >= 0);
}

```

At this point, the view is complete and we just have to create the controller. The controller will keep track of the view as well as the model. We will be handling events for the pressing of the Add button, Remove button, Hot List button as well as List Selection. Here is the basic framework for the controller:

```

import java.awt.event.*;
import javax.swing.*;

public class EmailBuddyApp extends JFrame implements DialogClientInterface {

    private EmailBuddyList    model;        // The model
    private EmailBuddyPanel    view;        // The view

```

```
// Here is the default constructor
public EmailBuddyApp(String title){
    super(title);

    // Initially, no buddies
    model = new EmailBuddyList();

    // Make a new viewing panel and add it to the pane
    view = new EmailBuddyPanel(model);
    getContentPane().add(view);

    // Make a listener for the add button
    view.getAddButton().addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            addBuddy();
        }
    });

    // Make a listener for the remove button
    view.getRemoveButton().addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event){
            removeBuddy();
        }
    });

    // Make a listener for the hot list checkbox
    view.getHotListButton().addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event){
            view.update();
        }
    });

    // Make a double-click listener
    view.getBuddyList().addMouseListener(new MouseAdapter() {
        public void mouseClicked(MouseEvent event){
            if (event.getClickCount() == 2)
                editBuddy();
            view.update();
        }
    });

    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize(600,300);
    setLocation(800,400);
}

...

// This is called when the user clicks the add button
private void addBuddy() { ... }

// This is called when the user clicks the remove button
private void removeBuddy() { ... }

// This is called when the user double clicks a buddy
private void editBuddy() { ... }

// Code that starts the application
public static void main(String[] args) {
    new EmailBuddyApp("Email Buddy Application").setVisible(true);
}
}
```

Notice that the code is straight forward. The hot list button event handler only requires a refreshing of the list, so the view's **update()** method is called.

Now let us examine the helper methods in turn. The **addBuddy()** method is called when the user clicks the **Add** button. This method should bring up the dialog box for adding a new buddy. We have not created this dialog box, but we will do so soon. The adding of the new email buddy should only occur if the user presses the OK button. If the CANCEL button is pressed, or the dialog box is closed down, then no email buddy should be added.

To make the code simpler, it is a good idea to create the new email buddy when the **Add** button is pressed so that we can pass this buddy into the dialog box so that its contents can be set. We can go ahead and add it to the model. Of course, if the user cancels the adding of the new buddy, we will have to remove this newly added buddy. In the **addBuddy()** event handler, we will just need to create the buddy, add it to the model and then open up the dialog box (which we have not yet created, but will do so soon):

```
private void addBuddy() {
    EmailBuddy aBuddy = new EmailBuddy();
    model.add(aBuddy);           // Add the buddy to the model

    // Now bring up the dialog box
    BuddyDetailsDialog dialog = new BuddyDetailsDialog(this,
        "New Buddy Details", true, aBuddy);
    dialog.setVisible(true);
}
```

This code will bring up the dialog box. Remember, that this main application implements the **DialogClientInterface**. Therefore, the **BuddyDetailsDialog** will be trying to call the **dialogFinished()** and **dialogCancelled()** methods, so we will need to write these:

```
// Called when dialog box is closed with OK button
public void dialogFinished() {
    view.update();
}
// Called when dialog box is closed with CANCEL button or manually closed
public void dialogCancelled() {
    // Remove the latest buddy that was added
    model.remove(model.getSize()-1);
}
```

The methods are simple. Since we added the buddy when in the **addBuddy()** method, when the user presses OK, there is nothing left to do except update the view. However, when the user presses CANCEL, we need to remove the last added buddy from the model ... which is done in the **dialogCancelled()** method here.

The **removeBuddy()** event handler is also easy to write. We just need to determine which buddy is selected from the list and then as long as there is someone selected ... we just remove the buddy by calling the model's **remove()** method and then update the view:

```
// This is called when the user clicks the remove button
private void removeBuddy() {
    int index = view.getBuddyList().getSelectedIndex();
    if (index >= 0) {
```

```

        model.remove(index);
        view.update();
    }
}

```

The **editBuddy()** event handler is a little more involved. First, we need to determine which buddy was selected ... we can grab the index in the list. Then, as long as there was a buddy selected, we can bring up the dialog box for that buddy. Unfortunately, however, the index in the list will be different from the index into the model's list if the hot list button is enabled. So we will need to handle these cases separately.

Likely, we will want to use the same dialog box as we did with the add button, but with the selected buddy's information as opposed to adding a new buddy. Here is the basic idea:

```

private void editBuddy() {
    EmailBuddy selectedBuddy;

    int    selectedIndex = view.getBuddyList().getSelectedIndex();
    if (selectedIndex >= 0) {
        if (view.getHotListButton().isSelected())
            selectedBuddy = model.getHotListBuddy(selectedIndex);
        else
            selectedBuddy = model.getBuddy(selectedIndex);
        if (selectedBuddy == null)
            return;
        BuddyDetailsDialog dialog = new BuddyDetailsDialog(this,
            "Edit Buddy Details", true, selectedBuddy);
        dialog.setVisible(true);
    }
}

```

Of course, there will be a problem when the user cancels the editing since the **dialogCancelled()** method removes the last buddy in the list! We do not want to do this removal if we are merely editing. To avoid this, we can add a new instance variable to the class to determine whether or not we are in "add" mode or "edit" mode:

```

// This is set to true if the dialog box was opened to add a new buddy
// and is set to false if it was opened to edit a buddy
private boolean    inAddMode;

```

Then, we can set this value to true in the **addBuddy()** event handler (i.e., `inAddMode = true;`) and false in the **editBuddy()** event handler (i.e., `inAddMode = false;`). The **dialogCancelled()** method will need to change as well:

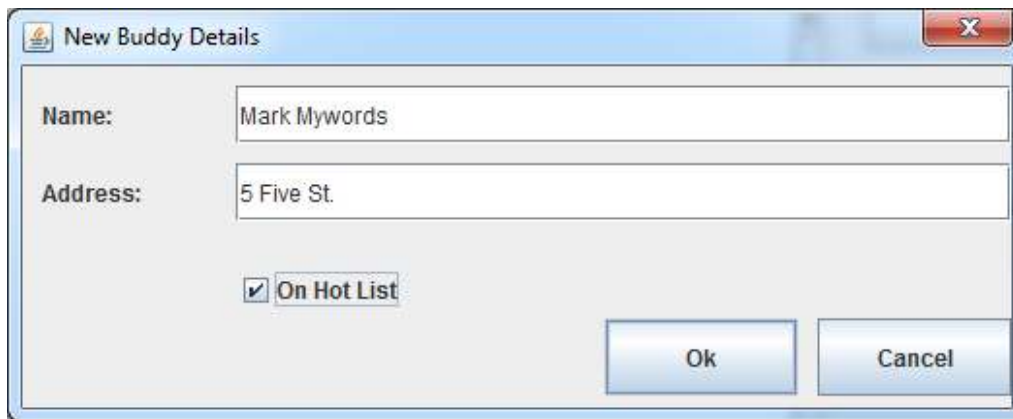
```

public void dialogCancelled() {
    // Remove the latest buddy that was added if in add mode
    if (inAddMode)
        model.remove(model.getSize()-1);
}

```

That is it! Now, we need to create the dialog box itself. It should allow the user to set the name, address and hot list status for the buddy that it is working on (i.e., either a newly added buddy or one being edited).

Here is what the dialog box will look like:



Here is the code:

```
import java.awt.event.*;
import javax.swing.*;

public class BuddyDetailsDialog extends JDialog {
    // This is a pointer to the email buddy that is being edited
    private EmailBuddy aBuddy;

    // These are the components of the dialog box
    private JLabel      aLabel;
    private JTextField  nameField;
    private JTextField  addressField;
    private JCheckBox   hotListButton;
    private JButton      okButton;
    private JButton      cancelButton;

    public BuddyDetailsDialog(JFrame owner, String title, boolean modal,
                               EmailBuddy bud) {
        super(owner, title, modal);
        aBuddy = bud;

        // Put all the components onto the window and given them initial values
        buildDialogWindow(aBuddy);

        // Add listeners for the Ok and Cancel buttons as well as window closing
        okButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                okButtonClicked();
            }
        });

        cancelButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                cancelButtonClicked();
            }
        });

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent event) {
                cancelButtonClicked();
            }
        });

        setSize(526, 214);
    }
}
```

```
        setLocationRelativeTo (owner);
    }

    private void buildDialogWindow (EmailBuddy aBuddy) {
        setLayout (null);

        // Add the name label
        aLabel = new JLabel ("Name:");
        aLabel.setLocation (10, 10);
        aLabel.setSize (80, 30);
        add (aLabel);
        // Add the name field
        nameField = new JTextField (aBuddy.getName ());
        nameField.setLocation (110, 10);
        nameField.setSize (400, 30);
        add (nameField);
        // Add the address label
        aLabel = new JLabel ("Address:");
        aLabel.setHorizontalAlignment (JLabel.LEFT);
        aLabel.setLocation (10, 50);
        aLabel.setSize (80, 30);
        add (aLabel);
        // Add the address field
        addressField = new JTextField (aBuddy.getAddress ());
        addressField.setLocation (110, 50);
        addressField.setSize (400, 30);
        add (addressField);
        // Add the onHotList button
        hotListButton = new JCheckBox ("On Hot List");
        hotListButton.setSelected (aBuddy.onHotList ());
        hotListButton.setLocation (110, 100);
        hotListButton.setSize (120, 30);
        add (hotListButton);
        // Add the Ok button
        okButton = new JButton ("Ok");
        okButton.setLocation (300, 130);
        okButton.setSize (100, 40);
        add (okButton);
        // Add the Cancel button
        cancelButton = new JButton ("Cancel");
        cancelButton.setLocation (410, 130);
        cancelButton.setSize (100, 40);
        add (cancelButton);
    }

    private void okButtonClicked () {
        aBuddy.setName (nameField.getText ());
        aBuddy.setAddress (addressField.getText ());
        aBuddy.onHotList (hotListButton.isSelected ());
        if (getOwner () != null)
            ((DialogClientInterface) getOwner ()).dialogFinished ();
        dispose ();
    }

    private void cancelButtonClicked () {
        if (getOwner () != null)
            ((DialogClientInterface) getOwner ()).dialogCancelled ();
        dispose ();
    }
}
```

This page was intentionally left blank.

Chapter 8

Abstract Data Types

What is in This Chapter ?

In this chapter we discuss the notion of Abstract Data Types (ADTs) as they pertain to storing collections of data in our programs. There are many common ADTs used in computer science. We will discuss here some of the common ones such as **Lists**, **Queues**, **Deque**, **Linked-Lists**, **Stacks**, **Sets** and **Dictionaries**. You will understand the differences between these various ADTs in terms of the operations that you can perform on them. Lastly, we will implement a Doubly-Linked Lists data structure to help you understand how pointers can be used to define a recursive data structure.



8.1 Common Abstract Data Types

Every time we define a new object, we are actually defining a new data type. That is, we are grouping attributes and behaviors to form a new type of data (i.e., object) we can use throughout our programs as if it were a single piece of data. There are actually some commonly used models for defining similar types of data:

*An **abstract data type (ADT)** is a mathematical model for a certain class of data structures that have similar behavior. (Wikipedia)*

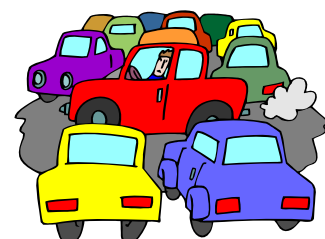
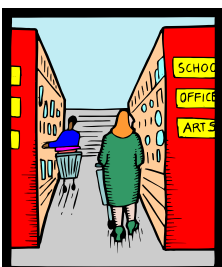
The word *abstract* here means that we are discussing data types in a general manner, without having a particular practical purpose or intention in mind. There are different types of ADTs, each with their own unique way for storing, accessing and modifying the data. Typically, an ADTs will store general data of any kind, although usually the data inside the ADT is all of the same kind ... or at least has something in common.

ADTs are a vital part of any programming language since they are used to *collect* data together in an "easy-to-use" way. We often use the term **collection** to represent these data types. There are advantages of using ADTs:

1. They help to simplify descriptions of abstract algorithms, thereby allowing us to write simplified pseudocode with less details (e.g., we can write pseudocode such as "**Add x to the list**" instead of "**Put x at position size in the array and then let size = size+1**").
2. They allow us to classify and evaluate data structures in regards to the common behaviors between data types (e.g., one ADT may have a more efficient remove operation while another may have a more efficient add or search operation. We could choose the ADT that best fits our needs).

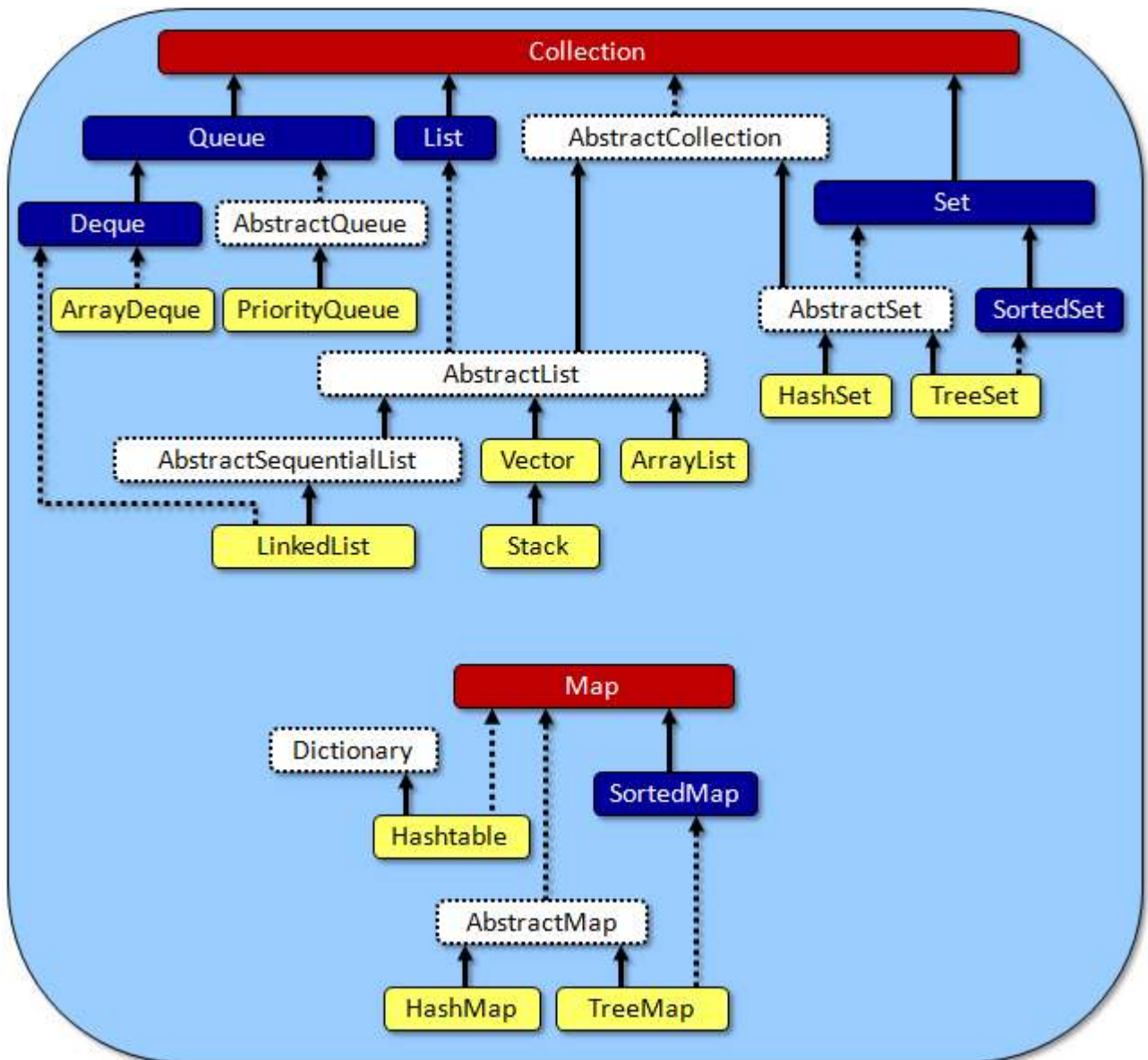
As we have already seen with **Arrays**, collections allow many objects to be collected together, stored and passed around as one object (i.e., the array itself is an object). Just about every useful application of any kind requires collections for situations such as:

- storing products on shelves in a store
- maintaining information on customers
- keeping track of cars for sale, for rental or for servicing
- a personal collection of books, CDs, DVDs, cards, etc...
- maintaining a shopping cart of items to be purchased from a website



In JAVA, there are a variety of ADT-related classes that can be used to represent these various programming needs. These ADTs are located in the `java.util.Collection` package, along with some other useful tools. In this set of notes, we investigate (very briefly) some of these JAVA collections in a way that will help a programmer understand which ADT is best for their particular programming application.

ADTs in JAVA are organized into a “seemingly complicated” hierarchy of JAVA interfaces and classes. There are two sub-hierarchies ... one is rooted at **Collection**, the other is rooted at **Map**. Here is a diagram showing part of this hierarchy:



In the above hierarchy, the red and dark blue represent java **interfaces**, the white classes represent **abstract** class and the yellow represent **concrete** classes. The solid arrows indicate **inheritance** while the dashes lines indicate that a class **implements** an interface. We will be discussing some of these classes in detail. You may want to refer back to this diagram once in a while to ensure that you understand how the classes differ and how they are similar.

Notice that there are 11 concrete classes, and that all of them indirectly implement the **Collection** or **Map** interface. Recall that an *interface* just specifies a list of method signatures ... not any code. That means, all of the concrete collection & map classes have something in common and that they all implement a common set of methods.

The main commonality between the collection classes is that they all store objects called their **elements**, which may be heterogeneous objects (i.e., the elements may be a mix of various (possibly unrelated) objects). Storing mixed kinds of objects in a **Collection** is allowed, but not often done unless there is something in common with the objects (i.e., they extend a common **abstract** class or implement a common **interface**).

The **Collection** interface defines common methods for querying (i.e., getting information from) and modifying (i.e., changing) the collection in some way. However, there are also various restrictions for each of the collection classes in terms of what they are allowed and not allowed to do when adding, removing and searching for data. We will look at the various classes that implement the **Collection** and **Map** interfaces.

It is not the purpose of this course to describe in-depth details on various kinds of collections and data structures. You will gain a deeper understanding of the advantages and disadvantages between data structures in your second year data structures course.

8.2 The List ADT

In real life, objects often appear in simple lists. For example, **Companies** may maintain a list of **Employees**, **Banks** may keep a list of **BankAccounts**, a **Person** may keep a list of "Things to Do". etc..

A **List** ADT allows us to access any of its elements at any time as well as insert or remove elements anywhere in the list at any time. The list will automatically shift the elements around in memory to make the needed room or reduce the unused space. The general **list** is the most flexible kind of list in terms of its capabilities.



*A **list** is an abstract data type that implements an ordered collection of values, where the same value may occur more than once.*

We use a general **List** whenever we have elements coming in and being removed in a random order. For example, when we have a shelf of library books, we may need to remove a book from anywhere on the shelf and we may insert books back into their proper location when they are returned.

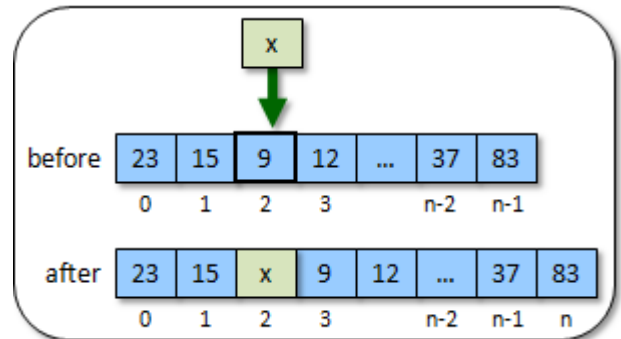
The elements in a general list are stored in a particular position in the list. As with arrays, elements in a general list are accessed according to their **index** position in the list.

The basic methods for inserting, removing, accessing and modifying items from a **List** are as follows:

add(int index, Object x)

Insert object **x** at position **index** in the list. Objects at positions **index + 1** through **n-1** move to the right by one position, increasing the size of the list by 1.

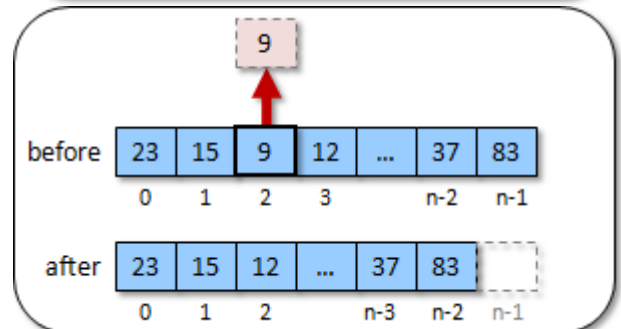
e.g., `aList.add(2, x)` will do this →



remove(int index)

Remove and return the object at position **index** in the list. Objects at positions **index + 1** through **n-1** move to the left by one position, decreasing the size of the list by 1.

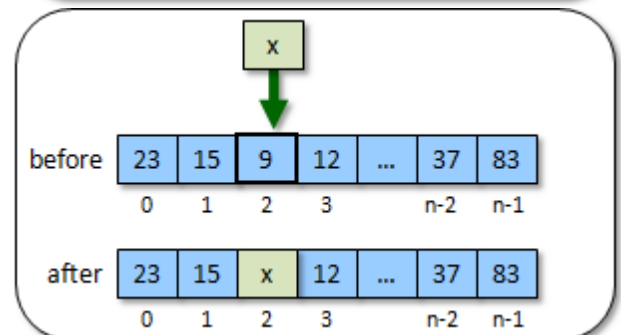
e.g., `aList.remove(2)` will return **9** →



set(int index, Object x)

Replace the object at position **index** in the list with the new object **x**. Objects at all other positions remain in their original position.

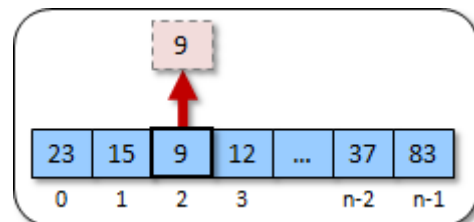
e.g., `aList.set(2, x)` will do this →



get(int index)

Return the object at position **index** in the list. The list is not changed in any way.

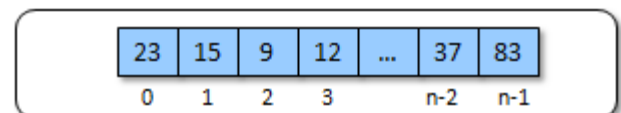
e.g., `x = aList.get(2)` will return **9** →



size()

Return the number of elements in the list.

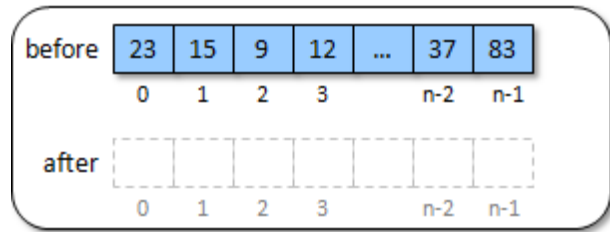
e.g., `n = aList.size()` will return **n** →



clear()

Remove all elements from the list.

e.g., `aList.clear()` will do this →

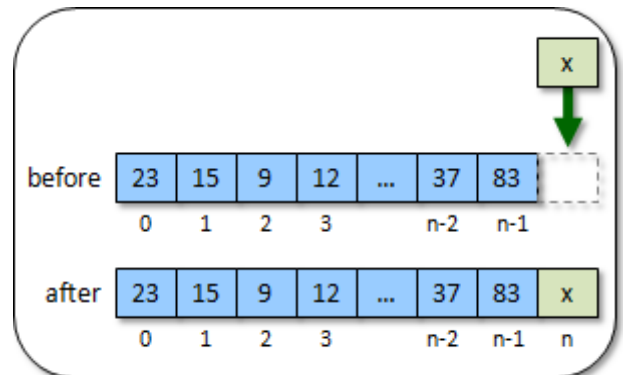


There are additional methods often available for convenience sake. Here are some:

add(Object x)

Insert object `x` at the end of the list, increasing the size of the list by 1.

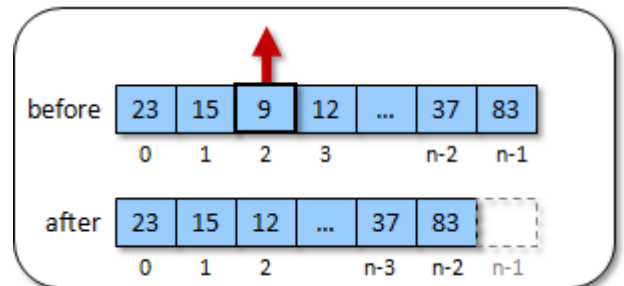
e.g., `aList.add(x)` will do this →



remove(Object x)

Remove the first occurrence of object `x` from the list. Assuming `x` was found at position `i`, then objects at positions `i + 1` through `n - 1` move to the left by one position, decreasing the size of the list by 1.

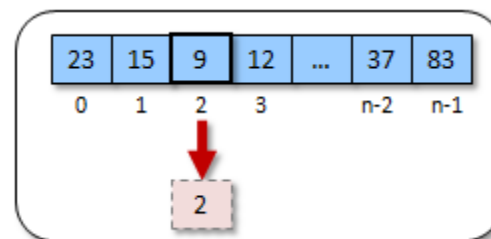
e.g., `aList.remove(9)` will do this →



indexOf(Object x)

Return the position of the first occurrence of object `x` in the list.

e.g., `i = aList.indexOf(9)` will return 2 →



isEmpty()

Return **true** if the number of elements in the list is 0, otherwise return **false**.

does the same as this:

```
return (aList.size() == 0);
```

contains(Object x)

Return **true** if `x` is contained in the list, otherwise return **false**.

does the same as this:

```
for (int i=0; i<aList.size(); i++)
    if (aList.get(i).equals(x))
        return true;
return false;
```

In JAVA, the *List* ADT is called an **ArrayList** and it is located in the **java.util** package, which must be imported in order to use this data type.

To create an **ArrayList**, we can simply call a constructor from the **ArrayList** class. Here is an example of creating an **ArrayList** and storing it in a variable so that we can use it:

```
ArrayList    myList;
myList = new ArrayList();
```

The above code allows us to store any kind of object in the **ArrayList**. We can then use the **ArrayList's add()** method to add objects to the end of the list in sequence as follows:

```
import java.util.ArrayList;

public class ArrayListTestProgram {
    public static void main(String[] args) {
        ArrayList    myList;

        myList = new ArrayList();
        myList.add("Hello");
        myList.add(25);
        myList.add(new Person());
        myList.add(new Truck());
        System.out.println(myList);
    }
}
```

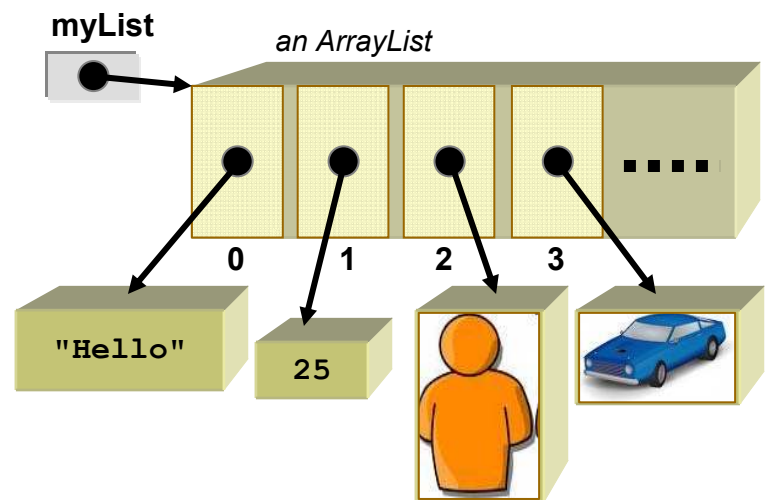
Notice in the above code that we are adding a **String**, an **int**, a **Person** object and a **Truck** object. Notice as well that at the top of the program we imported **java.util.ArrayList**. This is necessary in order for JAVA to know where to find the **ArrayList** class and its methods.

The output for the program is as follows (assuming that **Person** and **Truck** are defined classes that do not have **toString()** methods):

```
[Hello, 25, Person@addbf1, Truck@42e816]
```

Did you notice how **ArrayLists** look when you print them out? They show all the elements/items in the list separated by commas **,** in between square brackets **[]**.

This is the general format for an **ArrayList** that can hold any kinds of objects. However, it is "highly recommended" that we specify the *type* of objects that will be stored in the **ArrayList**. We do this by specifying the type between **<** and **>** characters just before the round brackets **()** as follows:



```
ArrayList<Object>    myList;
myList = new ArrayList<Object>();
```

If we know, for example, that all of the objects in the **ArrayList** will be **Strings** (e.g., names of people), then we should declare and create the list as follows:

```
ArrayList<String>    myList;
myList = new ArrayList<String>();
...
```

Similarly, if the objects to be stored in the list were of type **Person**, **BankAccount** or **Car** ... then we would specify the type as **<Person>**, **<BankAccount>** or **<Car>**, respectively.

Here is an example that uses the **get()** and **size()** methods:

```
ArrayList<Object>    myList;

myList = new ArrayList<Object>();
System.out.println(myList.size());    // outputs 0
myList.add("Hello");
myList.add(25);
myList.add(new Person());
myList.add(new Car());
System.out.println(myList.get(0));    // outputs "Hello"
System.out.println(myList.get(2));    // outputs Person@addbf1
System.out.println(myList.get(4));    // an IndexOutOfBoundsException
System.out.println(myList.size());    // outputs 4
```



Since Lists are perhaps the most commonly used data structure in computer science, we will do a couple of larger examples so that we get a full understanding of how to use them properly.

Example:



Consider a realistic use of the **ArrayList** object by creating classes called **Team** and **League** in which a **League** object will contain a bunch of **Team** objects. That is, the **League** object will have an instance variable of type **ArrayList** to hold onto the multiple **Team** objects within the league.



Consider first the creation of a **Team** class that will represent a single team in the league. For each team, we will maintain the team's **name** as well as the number of **wins**, **losses** and **ties** for the games that

they played. Here is the basic class (review the previous chapters in the notes if any of this is not clear):

```
public class Team {
    private String    name;        // The name of the Team
    private int       wins;        // The number of games that the Team won
    private int       losses;     // The number of games that the Team lost
    private int       ties;       // The number of games that the Team tied

    public Team(String aName) {
        this.name = aName;
        this.wins = 0;
        this.losses = 0;
        this.ties = 0;
    }

    // Get methods
    public String getName() { return name; }
    public int getWins() { return wins; }
    public int getLosses() { return losses; }
    public int getTies() { return ties; }

    // Modifying methods
    public void recordWin() { wins++; }
    public void recordLoss() { losses++; }
    public void recordTie() { ties++; }

    // Returns a text representation of a team
    public String toString() {
        return("The " + this.name + " have " + this.wins + " wins, " +
            this.losses + " losses and " + this.ties + " ties.");
    }

    // Returns the total number of points for the team
    public int totalPoints() {
        return (this.wins * 2 + this.ties);
    }

    // Returns the total number of games played by the team
    public int gamesPlayed() {
        return (this.wins + this.losses + this.ties);
    }
}
```

We can test out our **Team** object with the following test code, just to make sure it works:

```

public class TeamTestProgram {
    public static void main(String[] args) {
        Team    teamA, teamB;

        teamA = new Team("Ottawa Senators");
        teamB = new Team("Montreal Canadians");

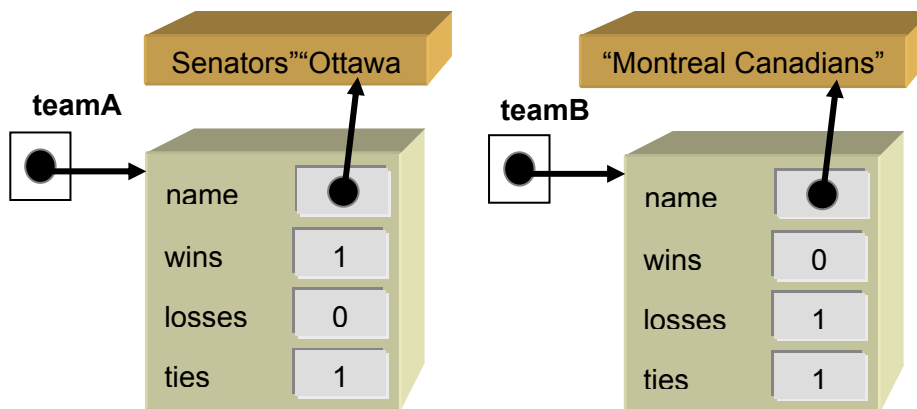
        // Simulate the playing of a game in which teamA beat teamB
        System.out.println(teamA.getName()+" just beat "+teamB.getName());
        teamA.recordWin();
        teamB.recordLoss();

        // Simulate the playing of another game in which they tied
        System.out.println(teamA.getName()+" just tied "+teamB.getName());
        teamA.recordTie();
        teamB.recordTie();

        //Now print out some statistics
        System.out.println(teamA);
        System.out.println(teamB);
        System.out.print("The " + teamA.getName() + " have ");
        System.out.print(teamA.totalPoints() + " points and played ");
        System.out.println(teamA.gamesPlayed() + " games.");
        System.out.print("The " + teamB.getName() + " have ");
        System.out.print(teamB.totalPoints() + " points and played ");
        System.out.println(teamB.gamesPlayed() + " games.");
    }
}

```

Here is what the **Team** objects look like after playing the two games:



Here is the output from our little test program:

```

Ottawa Senators just beat Montreal Canadians
Ottawa Senators just tied Montreal Canadians
The Ottawa Senators have 1 wins, 0 losses and 1 ties.
The Montreal Canadians have 0 wins, 1 losses and 1 ties.
The Ottawa Senators have 3 points and played 2 games.
The Montreal Canadians have 1 points and played 2 games.

```

Now let us implement the **League** class. A league will also have a **name** as well as an **ArrayList** (called **teams**) of **Team** objects. Here is the basic class structure (notice the **import** statement at the top):

```
import java.util.ArrayList;

public class League {
    private String          name;
    private ArrayList<Team> teams;

    public League(String n) {
        this.name = n;
        this.teams = new ArrayList<Team>(); // Doesn't make any Team objects
    }

    // This specifies the appearance of the League
    public String toString() {
        return ("The " + this.name + " league");
    }

    // Add the given team to the League
    public void addTeam(Team t) {
        teams.add(t);
    }
}
```

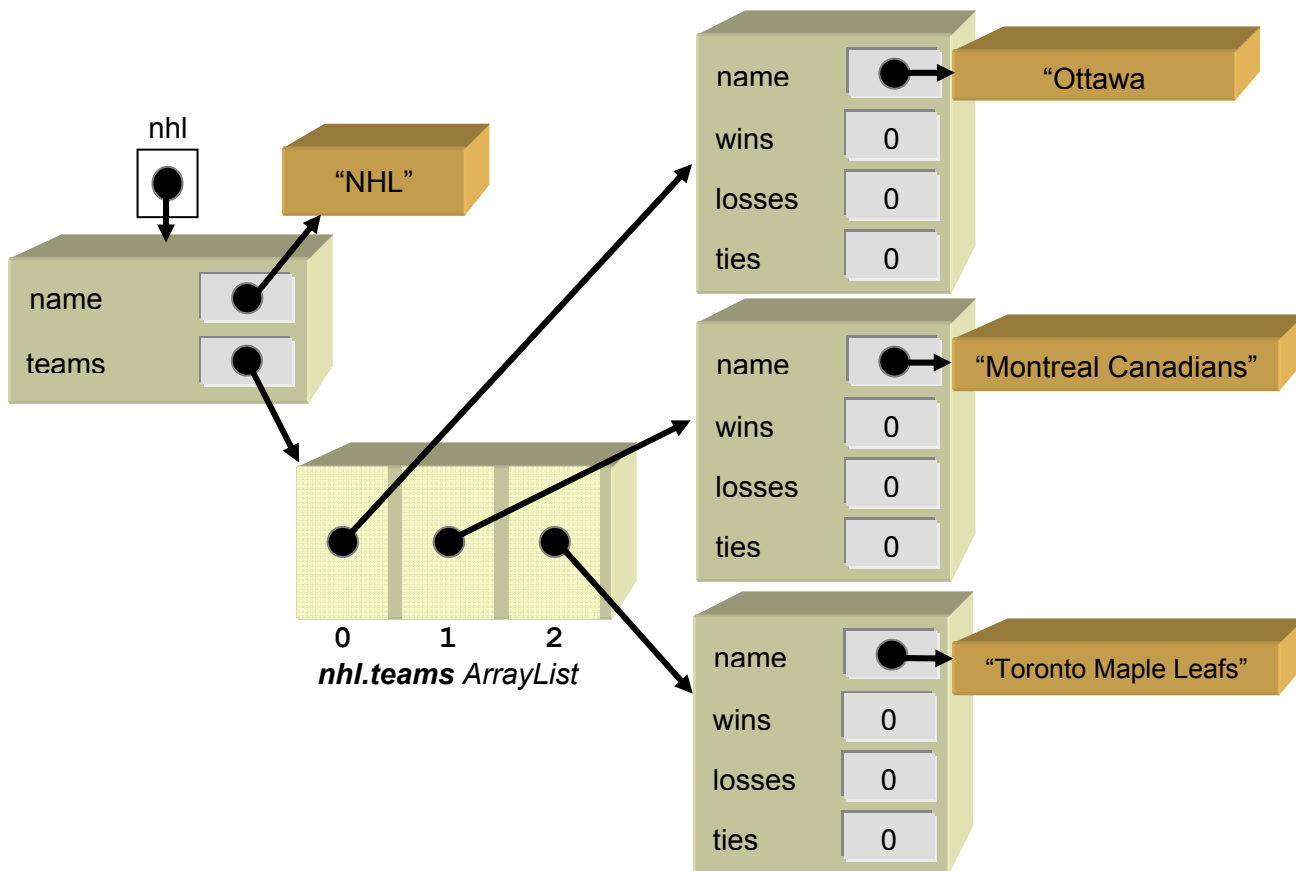
Notice that the **ArrayList** is created within the constructor and that it is initially empty. That means, a brand new league has no teams in it. It is important to note also that there are no **Team** objects created at this time.

At this point, we have defined two objects: **Team** and **League**. One thing that we will need to do is to be able to add teams to the league. Here is an example of how we can create a league with three teams in it:

```
League nhl;

nhl = new League("NHL");
nhl.addTeam(new Team("Ottawa Senators"));
nhl.addTeam(new Team("Montreal Canadiens"));
nhl.addTeam(new Team("Toronto Maple Leafs"));
```

In order to add the team to the league, we simply add it to the league's **teams** by using the **addTeam()** method which makes use of the **add()** method that is defined in the **ArrayList** class. Here is a diagram showing how the **League** object stores the 3 **Teams** ...



Suppose now that we wanted to print out the teams in the league. We will write a method in the **League** class called **showTeams()** to do this. The method will need to go through each team in the **teams** **ArrayList** and display the particular team's information ... perhaps using the **toString()** method from the **Team** class.

Hopefully, you "sense" that printing out all the teams involves repeating some code over and over again. That is, you should realize that we need a loop of some type. We have already discussed the **for** and **while** loops, but there is a special kind of **for** loop in JAVA that is to be used when traversing through a collection such as an **ArrayList**. This loop is called the "**for-each**" loop, and its structure is a little simpler than the traditional **for** loop. Below is how we can use the typical FOR loop as well as the "better" FOR-EACH loop to write the **showTeams()** method.

Using a Typical FOR Loop	Using a FOR-EACH Loop
<pre>public void showTeams() { for (int i=0; i<teams.size(); i++) { System.out.println(teams.get(i)); } }</pre>	<pre>public void showTeams() { for (Team t: teams) { System.out.println(t); } }</pre>

Notice that the **for-each** loop starts with **for** again, but this time the information within the round **()** brackets is different. The format of this information is as follows. First we specify the **type** of object that is in the **ArrayList** ... in this case **Team**. Then we specify a variable name

which will be used to represent the particular team as we loop through them all ... in this case we called it simply **t**.

Then we use a colon **:** character followed by the name of the ArrayList that we want to loop through ... in this case **teams**. So, if we were to *translate* the **for-each** loop into English, it would sound something like this: *"For each team t in the teams array list do the loop"*.

Notice that within the loop, we simply use **t** as we would use any other variable. In our example, **t** is the **Team** object that we are examining during that round through the loop. So **t** points to the 1st team in the league when we begin the loop, then it points to the 2nd team the next time through the loop, then the 3rd team etc..

Let us test our method out using the following test program:

```
public class LeagueTestProgram {
    public static void main(String[] args) {
        League nhl;

        nhl = new League("NHL");

        //Add a pile of teams to the league
        nhl.addTeam(new Team("Ottawa Senators"));
        nhl.addTeam(new Team("Montreal Canadiens"));
        nhl.addTeam(new Team("Toronto Maple Leafs"));
        nhl.addTeam(new Team("Vancouver Canucks"));
        nhl.addTeam(new Team("Edmonton Oilers"));
        nhl.addTeam(new Team("Washington Capitals"));
        nhl.addTeam(new Team("New Jersey Devils"));
        nhl.addTeam(new Team("Detroit Red Wings"));

        //Display the teams
        System.out.println("\nHere are the teams:");
        nhl.showTeams();
    }
}
```



Here is the output so far:

```
Here are the teams:
The Ottawa Senators have 0 wins, 0 losses and 0 ties.
The Montreal Canadiens have 0 wins, 0 losses and 0 ties.
The Toronto Maple Leafs have 0 wins, 0 losses and 0 ties.
The Vancouver Canucks have 0 wins, 0 losses and 0 ties.
The Edmonton Oilers have 0 wins, 0 losses and 0 ties.
The Washington Capitals have 0 wins, 0 losses and 0 ties.
The New Jersey Devils have 0 wins, 0 losses and 0 ties.
The Detroit Red Wings have 0 wins, 0 losses and 0 ties.
```

Notice that all the teams have no recorded wins, losses or ties. Lets write a method that will record a win and a loss for two teams that play together, and another method to record a tie when the two teams play and tie.

```

public void recordWinAndLoss (Team winner, Team loser) {
    winner.recordWin();
    loser.recordLoss();
}

public void recordTie (Team teamA, Team teamB) {
    teamA.recordTie();
    teamB.recordTie();
}

```



If we wanted to test these methods now, we could write test code like this:

```

League    nhl;
Team      team1, team2, team3;

nhl = new League("NHL");
nhl.addTeam(team1 = new Team("Ottawa Senators"));
nhl.addTeam(team2 = new Team("Montreal Canadians"));
nhl.addTeam(team3 = new Team("Toronto Maple Leafs"));

nhl.recordWinAndLoss(team1, team2);
nhl.recordTie(team1, team2);
nhl.recordWinAndLoss(team3, team2);
// ... etc ...

```

You should now notice something tedious. We would have to make variables for each team if we want to record wins, losses and ties among them. Why? Because the recording methods require **Team** objects ... the same **Team** objects that we added to the **League** ... so we would have to remember them ... hence requiring us to store them in a variable. Perhaps a better way to record wins, losses and ties would be to do something like this:

```

League    nhl;

nhl = new League("NHL");
nhl.addTeam(new Team("Ottawa Senators"));
nhl.addTeam(new Team("Montreal Canadians"));
nhl.addTeam(new Team("Toronto Maple Leafs"));

nhl.recordWinAndLoss("Ottawa Senators", "Montreal Canadians");
nhl.recordTie("Ottawa Senators", "Montreal Canadians");
nhl.recordWinAndLoss("Toronto Maple Leafs", "Montreal Canadians");

// ... etc ...

```

This way, we do not need to create extra variables. However, we would have to make new recording methods that took Strings (i.e., the **Team** names) as parameters instead of **Team** objects. Here are the methods that we would need to implement (notice the difference in the parameter types):

```
public void recordWinAndLoss (String winnerName, String loserName) {
    ...
}

public void recordTie (String teamAName, String teamBName) {
    ...
}
```

To make this work, however, we still need to get into the appropriate **Team** objects and update their wins/losses/ties. Therefore, we will have to take the incoming team names and find the **Team** objects that correspond with those names. We would need to do this 4 times: once for the **winnerName**, once for the **loserName**, once for **teamAName** and once for **teamBName**. Rather than repeat the code 4 times, we will make a method to do this particular sub-task of finding a team with a given name. Here is the method that we will write:

```
private Team teamWithName (String nameToLookFor) {
    Team answer;
    ...
    return answer;
}
```

Notice that it will take the team's name as a parameter and then return a **Team** object. How would we complete this method? We can use the **for-each** loop to traverse through all the teams and find the one with that name as follows:

```
private Team teamWithName (String nameToLookFor) {
    Team answer = null;
    for (Team t: teams) {
        if (t.name.equals(nameToLookFor))
            answer = t;
    }
    return answer;
}
```

Notice a few points. First, we set the answer to **null**. If we do not find a **Team** with the given name, the method returns **null** ... which is the only appropriate answer. Next, notice that for each team **t**, we compare its name with the incoming string **aName** and if these two strings are equal, then we have found the **Team** object that we want, so we store it in the **answer** variable to be returned at the completion of the loop.

This method can be shortened as follows:

```

private Team teamWithName(String nameToLookFor) {
    for (Team t: teams)
        if (t.getName().equals(nameToLookFor))
            return t;
    return null;
}

```

Now that this method has been created, we can use it in our methods for recording wins/losses and ties as follows:

```

public void recordWinAndLoss(String winnerName, String loserName) {
    Team winner, loser;

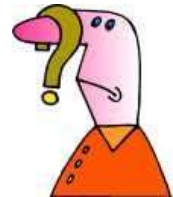
    winner = this.teamWithName(winnerName);
    loser = this.teamWithName(loserName);
    winner.recordWin();
    loser.recordLoss();
}

public void recordTie(String teamAName, String teamBName) {
    Team teamA, teamB;

    teamA = this.teamWithName(teamAName);
    teamB = this.teamWithName(teamBName);
    teamA.recordTie();
    teamB.recordTie();
}

```

The methods work as before, but there are potential problems. What if we cannot find the **Team** object with the given names (e.g., someone spelt the name wrong)? In this case, perhaps **winner**, **loser**, **teamA** or **teamB** will be **null** and we will get a **NullPointerException** when we try to access the team's attributes. We can check for this with an **if** statement.



```

public void recordWinAndLoss(String winnerName, String loserName) {
    Team winner, loser;

    winner = this.teamWithName(winnerName);
    loser = this.teamWithName(loserName);
    if ((winner != null) && (loser != null)) {
        winner.recordWin();
        loser.recordLoss();
    }
}

```



```

public void recordTie(String teamAName, String teamBName) {
    Team    teamA, teamB;

    teamA = this.teamWithName(teamAName);
    teamB = this.teamWithName(teamBName);
    if ((teamA != null) && (teamB != null)) {
        teamA.recordTie();
        teamB.recordTie();
    }
}

```

Now the games are only recorded when we have successfully identified the two **Team** objects that need to be updated as a result of the played game. Interestingly though, the same problem may occur in our previous recording methods ... that is ... the **Team** objects passed in may be **null**. Also, in our code, we already have method for recording the wins/losses/ties in the case where we have the **Team** objects, so we should call those methods from here. We can simply call the previous recording methods from these two new ones and move the **null**-checking in there instead as follows:

```

private Team teamWithName(String nameToLookFor) {
    for (Team t: teams)
        if (t.name.equals(nameToLookFor))
            return t;
    return null;
}

public void recordWinAndLoss(Team winner, Team loser) {
    if ((winner != null) && (loser != null)) {
        winner.recordWin();
        loser.recordLoss();
    }
}

public void recordTie(Team teamA, Team teamB) {
    if ((teamA != null) && (teamB != null)) {
        teamA.recordTie();
        teamB.recordTie();
    }
}

public void recordWinAndLoss(String winnerName, String loserName) {
    Team    winner, loser;

    winner = this.teamWithName(winnerName);
    loser = this.teamWithName(loserName);
    this.recordWinAndLoss(winner, loser);
}

public void recordTie(String teamAName, String teamBName) {
    Team    teamA, teamB;

    teamA = this.teamWithName(teamAName);
    teamB = this.teamWithName(teamBName);
    this.recordTie(teamA, teamB);
}

```



In fact, we can even shorten the last two methods by noticing that the variables are not really necessary:

```
public void recordWinAndLoss(String winnerName, String loserName) {
    this.recordWinAndLoss(this.teamWithName(winnerName),
                          this.teamWithName(loserName));
}

public void recordTie(String teamAName, String teamBName) {
    this.recordTie(this.teamWithName(teamAName),
                  this.teamWithName(teamBName));
}
```

Consider a method called **totalGamesPlayed()** which is supposed to return the total number of games played in the league. All we need to do is count the number of games played by all the teams (i.e., we will need some kind of counter) and then divide by 2 (since each game was played by two teams, hence counted twice). Here is the format:

```
public int totalGamesPlayed() {
    int total = 0;
    ...
    return total/2;
}
```



We will also need a **for-each** loop to go through each team:

```
public int totalGamesPlayed() {
    int total = 0;
    for (Team t: teams) {
        ...
    }
    return total/2;
}
```

Now, if you were to look back at the **Team** class, you would notice a method in there called **gamesPlayed()**. That means, we can ask a team how many games they played by simply calling that method. We should be able to make use of this value as follows:

```
public int totalGamesPlayed() {
    int total = 0;
    for (Team t: teams)
        total += t.gamesPlayed();

    return total/2;
}
```

Notice that the method is quite simple, as long as you break it down into simple steps like we just did. For more practice, let us find the team that is in first place (i.e., the **Team** object that has the most points). We can start again as follows:

```
public Team firstPlaceTeam() {
    Team teamWithMostPoints = null;
    ...
    return teamWithMostPoints;
}
```



Notice that it returns a **Team** object. Likely, you realize that we also need a **for-each** loop since we need to check all of the teams:

```
public Team firstPlaceTeam() {
    Team teamWithMostPoints = null;

    for (Team t: teams) {
        ...
    }
    return teamWithMostPoints;
}
```

Again, we can make use of a pre-defined method in the **Team** class called **totalPoints()** which returns the number of points for a particular team:

```
public Team firstPlaceTeam() {
    int points;
    Team teamWithMostPoints = null;

    for (Team t: teams) {
        points = t.totalPoints();
    }
    return teamWithMostPoints;
}
```

But now what do we do? The current code will simply grab each team's point values one at a time. We need to somehow compare them. Many students have trouble breaking this problem down into simple steps. The natural tendency is to say to yourself "I will compare the 1st team's points with the 2nd team's points and see which is greater". If we do this however, then what do we do with that answer? How does the third team come into the picture?

Hopefully, after some thinking, you would realize that as we traverse through the teams, we need to keep track of (i.e., remember) the best one so far.

Imagine for example, searching through a basket of apples to find the best one. Would you not grab an apple and hold it in your hand and then look through the

other apples and compare them with the one you are holding in your hand? If you found a better one, you would simply trade the one currently in your hand with the new better one. By the time you reach the end of the basket, you are holding the best apple.



Well we are going to do the same thing. The **teamWithMostPoints** variable will be like our good apple that we are holding. Whenever we find a team that is better (i.e., more points) than this one, then that one becomes the **teamWithMostPoints**. Here is the code:

```
public Team firstPlaceTeam() {
    Team teamWithMostPoints = null;

    for (Team t: teams) {
        if (t.totalPoints() > teamWithMostPoints.totalPoints())
            teamWithMostPoints = t;
    }
    return teamWithMostPoints;
}
```

Does it make sense? There is one small issue though. Just like we need to begin our apple checking by picking up a first apple, we also need to pick a team (any **Team** object) to be the “best” one before we start the search. Currently the **teamWithMostPoints** starts off at **null** so we need to set this to a valid **Team** so start off. We can perhaps take the first **Team** in the **teams** **ArrayList**:

```
public Team firstPlaceTeam() {
    Team teamWithMostPoints = teams.get(0);

    for (Team t: teams) {
        if (t.totalPoints() > teamWithMostPoints.totalPoints())
            teamWithMostPoints = t;
    }
    return teamWithMostPoints;
}
```

We are not done yet! It is possible, in a weird scenario, that there are no teams in the league! In this case **teams.get(0)** will return **null** and we will get a **NullPointerException** again when we go to ask for the **totalPoints()**.

So, we would need to add a special case to return **null** if the **teams** list is empty. Here is the new code ...

```

public Team firstPlaceTeam() {
    Team teamWithMostPoints;

    if (teams.size() == 0)
        return null;

    teamWithMostPoints = teams.get(0);
    for (Team t: teams) {
        if (t.totalPoints() > teamWithMostPoints.totalPoints())
            teamWithMostPoints = t;
    }
    return teamWithMostPoints;
}

```

What would we change in the above code if we wanted to write a method called **lastPlaceTeam()** that returned the team with the least number of points? Try to do it.

How could we write a method called **undefeatedTeams()** that returned an **ArrayList<Team>** of all teams that have never lost a game? We would begin by specifying the proper return type:

```

public ArrayList<Team> undefeatedTeams() {
    ArrayList<Team> undefeated = new ArrayList<Team>();

    for (Team t: teams) {
        ...
    }
    return undefeated;
}

```

Now we would check each team that has not lost any games and add them to the result list:

```

public ArrayList<Team> undefeatedTeams() {
    ArrayList<Team> undefeated = new ArrayList<Team>();

    for (Team t: teams) {
        if (t.getLosses() == 0)
            undefeated.add(t);
    }
    return undefeated;
}

```

Another interesting method would be one that removes all teams from the league that have never won a game. Intuitively, here is what we may do:

```
public void removeLosingTeams() {
    for (Team t: teams) {
        if (t.getWins() == 0)
            teams.remove(t);
    }
}
```



However, this code will not work since it will generate a **ConcurrentModificationException** in JAVA. That is, we need to be careful not to remove items from a list that we are iterating through. As it turns out, the FOR EACH loop does not allow us to remove while iterating through the list. Using a standard FOR loop, however, we can make it work. The following code "almost works" ... in that it does not produce an exception ... but something is still wrong.

```
public void removeLosingTeams() {
    for (int i=0; i<teams.size(); i++) {
        Team t = teams.get(i);
        if (t.getWins() == 0)
            teams.remove(t);
    }
}
```

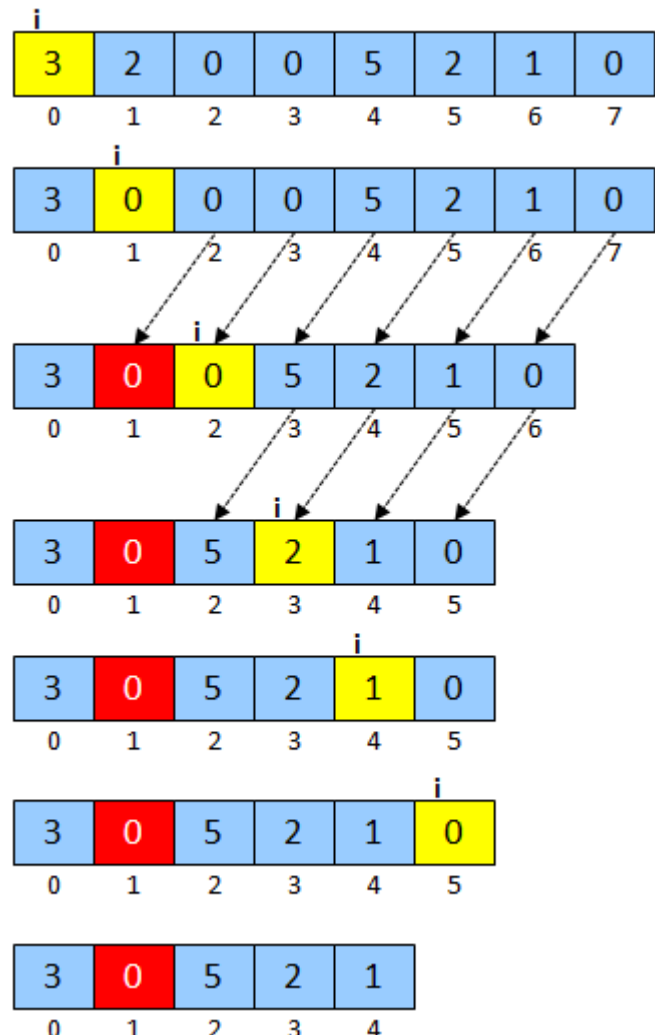


The code above may not remove all teams that have no wins. Why not? Consider what happens if two teams in a row have no wins.

In the picture to the right, imagine that these values represent the number of wins for the teams in the **teams** list. Notice how the index **i** is moved along through the loop as shown by the yellow square. When the team at position 1 is encountered, it has no wins, so it is removed ... and all other teams are moved back one position in the list.

Then, we continue with the loop as usual. However, the next time through the loop the **i** has moved to position 2. However, a 0-win team (shown red) has just been shifted into position 1 but never checked. Thus, by the time the loop has ended we never checked the team in position 1 and therefore it remains in the list.

How can we fix this? It is simple. Just ensure that we do not move the index to the next position in the case that we are doing a remove operation. We can accomplish this by



subtracting **1** from the index **i**, so that when the **for** loop increments **i**, it cancels out the increment and **i** remains in the same position.

```
public void removeLosingTeams() {
    for (int i=0; i<teams.size(); i++) {
        Team t = teams.get(i);
        if (t.getWins() == 0) {
            teams.remove(t);
            i--;
        }
    }
}
```



Now the code should work properly. Here is a program that can be used to test our methods:

```
public class LeagueTestProgram2 {
    public static void main(String[] args) {
        League nhl = new League("NHL");

        // Add a pile of teams to the league
        nhl.addTeam(new Team("Ottawa Senators"));
        nhl.addTeam(new Team("Montreal Canadiens"));
        nhl.addTeam(new Team("Toronto Maple Leafs"));
        nhl.addTeam(new Team("Vancouver Canucks"));
        nhl.addTeam(new Team("Edmonton Oilers"));
        nhl.addTeam(new Team("Washington Capitals"));
        nhl.addTeam(new Team("New Jersey Devils"));
        nhl.addTeam(new Team("Detroit Red Wings"));

        // Now we will record some games
        nhl.recordWinAndLoss("Ottawa Senators", "New Jersey Devils");
        nhl.recordWinAndLoss("Edmonton Oilers", "Montreal Canadiens");
        nhl.recordTie("Ottawa Senators", "Detroit Red Wings");
        nhl.recordWinAndLoss("Montreal Canadiens", "Washington Capitals");
        nhl.recordWinAndLoss("Ottawa Senators", "Edmonton Oilers");
        nhl.recordTie("Washington Capitals", "Edmonton Oilers");
        nhl.recordTie("Detroit Red Wings", "New Jersey Devils");
        nhl.recordWinAndLoss("Vancouver Canucks", "Toronto Maple Leafs");
        nhl.recordWinAndLoss("Toronto Maple Leafs", "Edmonton Oilers");
        nhl.recordWinAndLoss("New Jersey Devils", "Detroit Red Wings");

        // This one will not work
        nhl.recordWinAndLoss("Mark's Team", "Detroit Red Wings");

        // Now display the teams again
        System.out.println("\nHere are the teams after recording the " +
            "wins, losses and ties:\n");
        nhl.showTeams();
    }
}
```

```

// Here are some statistics
System.out.println("\nThe total number of games played is " +
    nhl.totalGamesPlayed());
System.out.println("The first place team is " +
    nhl.firstPlaceTeam());
System.out.println("The last place team is " +
    nhl.lastPlaceTeam());
System.out.println("The undefeated teams are " +
    nhl.undefeatedTeams());
System.out.println("Removing teams that never won ... ");
nhl.removeLosingTeams();
System.out.println("The teams are: ");
nhl.showTeams();
}
}

```

Here would be the output (make sure that it makes sense to you) ...

Here are the teams after recording the wins, losses and ties:

```

The Ottawa Senators have 2 wins, 0 losses and 1 ties.
The Montreal Canadians have 1 wins, 1 losses and 0 ties.
The Toronto Maple Leafs have 1 wins, 1 losses and 0 ties.
The Vancouver Cannucks have 1 wins, 0 losses and 0 ties.
The Edmonton Oilers have 1 wins, 2 losses and 1 ties.
The Washington Capitals have 0 wins, 1 losses and 1 ties.
The New Jersey Devils have 1 wins, 1 losses and 1 ties.
The Detroit Red Wings have 0 wins, 1 losses and 2 ties.

```

```

The total number of games played is 10
The first place team is The Ottawa Senators have 2 wins, 0 losses and 1 ties.
The last place team is The Washington Capitals have 0 wins, 1 losses and 1 ties.
The undefeated teams are [The Ottawa Senators have 2 wins, 0 losses and 1 ties.,
The Vancouver Cannucks have 1 wins, 0 losses and 0 ties.]
Removing teams that never won ...
The teams are:
The Ottawa Senators have 2 wins, 0 losses and 1 ties.
The Montreal Canadians have 1 wins, 1 losses and 0 ties.
The Toronto Maple Leafs have 1 wins, 1 losses and 0 ties.
The Vancouver Cannucks have 1 wins, 0 losses and 0 ties.
The Edmonton Oilers have 1 wins, 2 losses and 1 ties.
The New Jersey Devils have 1 wins, 1 losses and 1 ties.

```

Supplemental Information

There is an additional class called **Vector** which has the same functionality as the **ArrayList** class. In fact, in most situations, you can simply replace the word **ArrayList** by **Vector** and your code will still compile. There is a small **difference** between **ArrayLists** and **Vectors**. They have the same functionality, but **ArrayLists** are faster because they have methods that are not **synchronized**. **Vectors** allow multiple processes (or multiple "programs") to access/modify them at the same time, so they have extra code in the methods to ensure that the **Vector** is shared properly and safely between the processes. We will not talk any more about this in this course. You should always use **ArrayLists** when creating simple programs.

8.3 The Queue ADT

Consider the **Queue** ADT.

*A **queue** is an abstract data type that stores elements in a first-in-first-out order. Elements are added at one end and removed from the other.*

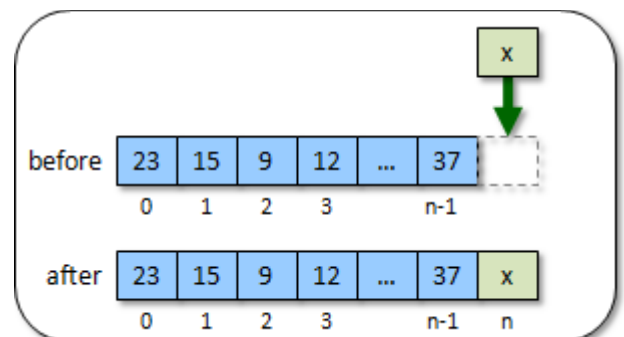


Hence, the first element to be added to the queue is the first element to be taken out of the queue. This is analogous to a line-up that we see every day. The first person in line is the first person served (i.e., first-come-first-served). When people arrive, they go to the back of the line. People get served from the front of the line first. Therefore, with a queue, we add to the back and remove from the front. We are not allowed to insert or remove elements from the middle of the queue. Why is this restriction a good idea? Well, depending on how the queue is implemented, it can be more efficient (i.e., faster) to insert and remove elements since we know that all such changes will occur at the front or back of the queue. Removing from the front may then simply require moving the “front-of-the-line pointer” instead of shifting elements over. Also, adding to the back may require extending the “back-of-the-line pointer”. Typical methods for **Queues** are:

add(Object x)

Insert object **x** at the end of the queue. This operation is sometimes called **push(x)** or **enqueue(x)**.

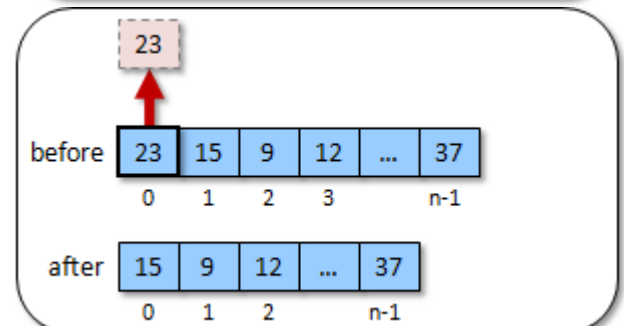
e.g., `aQueue.add(x)` will do this →



remove()

Remove and return the object at the front of the queue. The next item in the queue becomes the front item. This operation is sometimes called **pop()** or **dequeue()**.

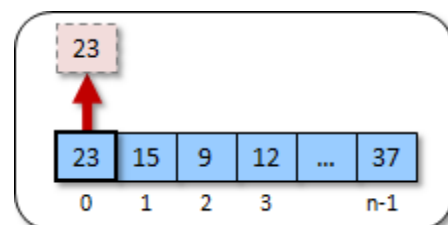
e.g., `x = aQueue.remove()` will return **23** →



peek()

Return (but do not remove) the object at the front of the queue. This operation is sometimes called **front()**.

e.g., `x = aQueue.peek()` will do this →



`size()`, `isEmpty()` and `clear()`

Does the same as with lists

Another more specialized type of queue is the **PriorityQueue**:

*A **PriorityQueue** is a queue in which the elements also maintain a priority.*

That is, we still add to the back of the queue and remove from the front, but elements with higher priority are automatically shifted closer to the front before lower priority elements.

As a real life example, when we go to the hospital for an “emergency”, we wait in line (6 to 8 hours typically). We normally get served in the order that we came in at. However, if someone comes in after us who is bleeding or unconscious, they automatically get bumped up ahead of us since their injuries are likely more serious and demand immediate attention. We may think of a **PriorityQueue** as a **sorted** queue.



The **PriorityQueue** is used in the same way as a regular **Queue**, except that we must ensure that each element added to the queue is given a priority. Hence, we are sometimes required to specify the priority of an item when we add it to the queue:

`add(int priority, Object x)`

However, in JAVA, we simply include the priority of an object as part of the object itself, and so we just use `add(Object x)` for priority queues in JAVA.

Example:

Consider the following **Person** class definition:

```
public class Person {
    private String    name;
    private int       age;

    public Person(String n, int a) {
        name = n;
        age = a;
    }

    public int getAge() { return age; }
    public String getName() { return name; }

    public String toString() {
        return age + " year old " + name;
    }
}
```

How could we simulate some people getting in a lineup and being served on a first-come-first-served basis? We can use a **Queue** to do this. In JAVA, there are a few different

implementations of the **Queue** ADT. We will use the one called **ArrayBlockingQueue** which is in the **java.util.concurrent** package. Here is a simple test program that creates a lineup and then serves the people one at a time by taking the first person in line each time. Notice how the code is arranged so that a nice log output is obtained that we can follow along with to see if our program does what it is supposed to be doing.

```
import java.util.concurrent.ArrayBlockingQueue;

public class QueueTestProgram {
    public static void main(String[] args) {
        ArrayBlockingQueue<Person>    lineup;
        lineup = new ArrayBlockingQueue<Person>(10);

        System.out.print("Adding three customers to the line ... ");
        lineup.add(new Person("Bob", 12));
        lineup.add(new Person("Mary", 6));
        lineup.add(new Person("Steve", 10));

        System.out.println("Here is who is in line at the moment:");
        System.out.println(lineup);

        System.out.print("Serving next customer... ");
        System.out.println(lineup.remove());
        System.out.println("Here is who is in line now:");
        System.out.println(lineup);

        System.out.print("Serving another customer ... ");
        System.out.println(lineup.remove());
        System.out.println("Here is who is in line now:");
        System.out.println(lineup);

        System.out.print("Adding three more customers to the line ... ");
        lineup.add(new Person("Ralph", 16));
        lineup.add(new Person("Jen", 13));
        lineup.add(new Person("Max", 18));
        System.out.println("Here is who is in line now:");
        System.out.println(lineup);

        System.out.print("Serving next customer ... ");
        System.out.println(lineup.remove());
        System.out.println("Here is who is in line now:");
        System.out.println(lineup);

        System.out.print("Adding four customers to the line ... ");
        lineup.add(new Person("Dave", 4));
        lineup.add(new Person("Sam", 17));
        lineup.add(new Person("Lyn", 8));
        lineup.add(new Person("Betty", 9));
        System.out.println("Here is who is in line now:");
        System.out.println(lineup);
        System.out.print("Here is who is at the front of the line ...");
        System.out.println(lineup.peek());
        System.out.print("Serving next customer ... ");
        System.out.println(lineup.remove());
        System.out.println("Here is who remains in the line:");
        System.out.println(lineup);
        System.out.println("Serving all remaining customers ... ");
    }
}
```

```

        while(!lineup.isEmpty()) {
            System.out.print("Serving next customer ... ");
            System.out.println(lineup.remove());
        }
        System.out.println("Here is who remains in the line:");
        System.out.println(lineup);
    }
}

```

Here is the output that is produced:

```

Adding three customers to the line ... Here is who is in line at the moment:
[12 year old Bob, 6 year old Mary, 10 year old Steve]
Serving next customer ... 12 year old Bob
Here is who is in line now:
[6 year old Mary, 10 year old Steve]
Serving another customer ... 6 year old Mary
Here is who is in line now:
[10 year old Steve]
Adding three more customers to the line ... Here is who is in line now:
[10 year old Steve, 16 year old Ralph, 13 year old Jen, 18 year old Max]
Serving next customer ... 10 year old Steve
Here is who is in line now:
[16 year old Ralph, 13 year old Jen, 18 year old Max]
Adding four customers to the line ... Here is who is in line now:
[16 year old Ralph, 13 year old Jen, 18 year old Max, 4 year old Dave, 17
year old Sam, 8 year old Lyn, 9 year old Betty]
Here is who is at the front of the line ...16 year old Ralph
Serving next customer ... 16 year old Ralph
Here is who remains in the line:
[13 year old Jen, 18 year old Max, 4 year old Dave, 17 year old Sam, 8 year
old Lyn, 9 year old Betty]
Serving all remaining customers ...
Serving next customer ... 13 year old Jen
Serving next customer ... 18 year old Max
Serving next customer ... 4 year old Dave
Serving next customer ... 17 year old Sam
Serving next customer ... 8 year old Lyn
Serving next customer ... 9 year old Betty
Here is who remains in the line:
[]

```

Example:

In a **PriorityQueue**, when we add items, they usually get shuffled around inside according to their priority. Therefore, we may not necessarily know the order of the items afterwards ... except that they will be in some sort of prioritized order.

Consider the **QueueTestProgram**. We can change the **ArrayBlockingQueue** to **PriorityQueue** to have a prioritized queue for our people. However, if we were to run the code, we would get an exception:

```
java.lang.ClassCastException: Person cannot be cast to java.lang.Comparable
```

The problem is that JAVA does not know how to compare **Person** objects in order to be able to sort them. It is telling us that **Person** must implement the **Comparable** interface. Instead of supplying a priority when we add the objects to the queue, the items are sorted by means of a **Comparable** interface. That means, each object that we store in the **PriorityQueue**, must implement methods **compareTo()** ... which are used determine the sort order (i.e., priority).

So, we should add **implements Comparable<Person>** to the **Person** class definition:

```
public class Person implements Comparable<Person> {
    ...
}
```

Interestingly, the additional **<Person>** at the end of **Comparable** indicates to JAVA that we will only be comparing **Person** objects, not **Person** objects with other types of objects.

But how do we write a **compareTo()** method ? It takes a single object parameter:

```
public int compareTo(Person p) { ... }
```

The method returns an **int**. This integer reflects the ordering between the receiver and the parameter. If a negative value is returned from the method, this informs JAVA that the receiver has higher priority (i.e., comes before in the ordering) than the incoming parameter object. Likewise, a positive value indicates lower priority and a zero value indicates that they are equal priority.



Lets now give it a try for **Person** objects. If we want to prioritize by means of their increasing ages (i.e., younger first), this would be the **compareTo()** method:

```
public int compareTo(Person p) {
    return (this.age - p.age);
}
```

Assume now that we ran the following program:

```
import java.util.PriorityQueue;

public class PriorityQueueTestProgram {
    public static void main(String[] args) {
        PriorityQueue<Person> lineup = new PriorityQueue<Person>(10);

        System.out.print("Adding 10 customers to the line ... ");
        lineup.add(new Person("Bob", 12));
        lineup.add(new Person("Mary", 6));
        lineup.add(new Person("Steve", 10));
        lineup.add(new Person("Ralph", 16));
        lineup.add(new Person("Jen", 13));
        lineup.add(new Person("Max", 18));
        lineup.add(new Person("Dave", 4));
        lineup.add(new Person("Sam", 17));
        lineup.add(new Person("Lyn", 8));
        lineup.add(new Person("Betty", 9));
    }
}
```

```

System.out.println("Here is who is in line now:");
System.out.println(lineup);
System.out.print("Here is who is at the front of the line ...");
System.out.println(lineup.peek());

System.out.println("Serving all customers ... ");
while(!lineup.isEmpty()) {
    System.out.print("Serving next customer ... ");
    System.out.println(lineup.remove());
}
System.out.println("Here is who remains in the line:");
System.out.println(lineup);
}
}

```

Interestingly, the output is as follows:

```

Adding 10 customers to the line ... Here is who is in line now:
[4 year old Dave, 8 year old Lyn, 6 year old Mary, 12 year old Bob, 9 year
old Betty, 18 year old Max, 10 year old Steve, 17 year old Sam, 16 year old
Ralph, 13 year old Jen]
Here is who is at the front of the line ...4 year old Dave
Serving all customers ...
Serving next customer ... 4 year old Dave
Serving next customer ... 6 year old Mary
Serving next customer ... 8 year old Lyn
Serving next customer ... 9 year old Betty
Serving next customer ... 10 year old Steve
Serving next customer ... 12 year old Bob
Serving next customer ... 13 year old Jen
Serving next customer ... 16 year old Ralph
Serving next customer ... 17 year old Sam
Serving next customer ... 18 year old Max
Here is who remains in the line:
[]

```



Notice that the items in the queue do not seem sorted at all ! That is because a **PriorityQueue** does not actually sort the items, it simple makes sure that the item at the front of the queue is the one with highest priority. In this case, that is the youngest person ... which is indeed at the front of the queue. To get the items in sorted order, we simply extract them from the queue one at a time as shown in the **while** loop from the code above. Indeed, you can see that as we extract the items one at a time, they come out in properly prioritized order.

What if we wanted to prioritize the people by their last names instead ? To do this, we would need to alter the **compareTo()** method to compare names, not ages. Lets make a subclass of **Person** called **AlphaPerson** that has a different **compareTo()** method:

```

public class AlphaPerson extends Person {
    public AlphaPerson(String n, int a) { super(n,a); }
    // Used to compare Persons by alphabetical order of last names
    public int compareTo(Person p) {
        return this.name.compareTo(p.name); // assumes that name
                                                // is declared protected
                                                // in the Person class
    }
}

```

Notice that the parameter for the `compareTo()` is **Person**, not **AlphaPerson**. This is because **Person** implements the **Comparable<Person>** interface, which specifies type **Person** and **AlphaPerson** inherits from **Person**. We are not allowed to implement both **Comparable<Person>** and **Comparable<AlphaPerson>** in the same class. Consider the output then from the following program ...

```
import java.util.PriorityQueue;
public class PriorityQueueTestProgram2 {
    public static void main(String[] args) {
        PriorityQueue<AlphaPerson> lineup = new PriorityQueue<AlphaPerson>(10);

        System.out.print("Adding 10 customers to the line ... ");
        lineup.add(new AlphaPerson("Bob", 12));
        lineup.add(new AlphaPerson("Mary", 6));
        lineup.add(new AlphaPerson("Steve", 10));
        lineup.add(new AlphaPerson("Ralph", 16));
        lineup.add(new AlphaPerson("Jen", 13));
        lineup.add(new AlphaPerson("Max", 18));
        lineup.add(new AlphaPerson("Dave", 4));
        lineup.add(new AlphaPerson("Sam", 17));
        lineup.add(new AlphaPerson("Lyn", 8));
        lineup.add(new AlphaPerson("Betty", 9));
        System.out.println("Here is who is in line now:");
        System.out.println(lineup);
        System.out.print("Here is who is at the front of the line ...");
        System.out.println(lineup.peek());

        System.out.println("Serving all customers ... ");
        while(!lineup.isEmpty()) {
            System.out.print("Serving next customer ... ");
            System.out.println(lineup.remove());
        }
        System.out.println("Here is who remains in the line:");
        System.out.println(lineup);
    }
}
```

Here is the output ... notice how the people are removed in alphabetical order of their name:

```
Adding 10 customers to the line ... Here is who is in line now:
[9 year old Betty, 12 year old Bob, 4 year old Dave, 8 year old Lyn, 13 year old
Jen, 10 year old Steve, 18 year old Max, 17 year old Sam, 16 year old Ralph, 6
year old Mary]
Here is who is at the front of the line ...9 year old Betty
Serving all customers ...
Serving next customer ... 9 year old Betty
Serving next customer ... 12 year old Bob
Serving next customer ... 4 year old Dave
Serving next customer ... 13 year old Jen
Serving next customer ... 8 year old Lyn
Serving next customer ... 6 year old Mary
Serving next customer ... 18 year old Max
Serving next customer ... 16 year old Ralph
Serving next customer ... 17 year old Sam
Serving next customer ... 10 year old Steve
Here is who remains in the line:
[]
```



So, we can prioritize the items in any way by creating an appropriate `compareTo()` method.

8.4 The Deque ADT

Consider the **Deque** ADT:

A **deque** is an abstract data type that is analogous to a **double-ended queue**. Elements are added/removed to/from either end.

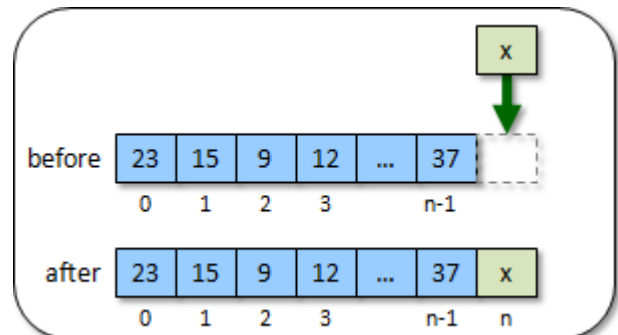
A deque allows us to add/remove from the front or the back of the queue at any time, but no modifications to the middle. It has the same advantages of a regular single-ended queue, but is a little more flexible in that it allows removal from the back of the queue and insertion at the front. An example of where we might use a deque is when we implement “Undo” operations in a piece of software. Each time we do an operation, we **add** it to the front of the deque. When we do an undo, we **remove** it from the front of the deque. Since undo operations usually have a fixed limit defined somewhere in the options (i.e., maximum 20 levels of undo), we remove from the back of the deque when the limit is reached. Typical methods for **Deques** are:



addFirst(Object x) and **addLast(Object x)**

Insert object **x** at the front (or back) of the deque.

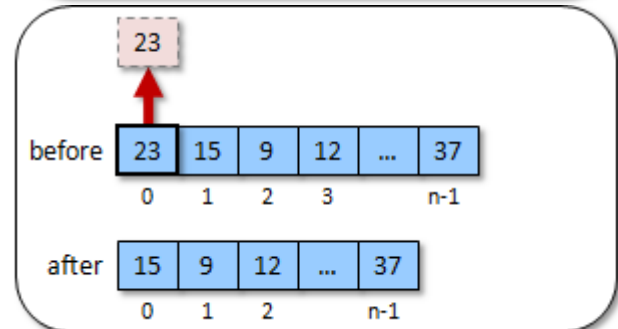
e.g., `aDeque.addLast(x)` will do this →



removeFirst() and **removeLast()**

Remove and return the object at the front (or back) of the deque. The next item in the deque becomes the front (or back) item.

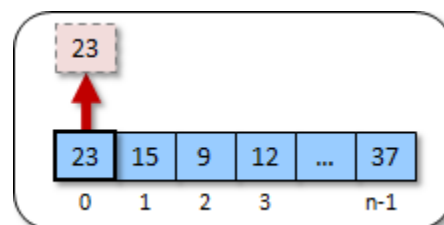
e.g., `x = aDeque.removeFirst()` will return **23** →



peekFirst() and **peekLast()**

Return (but do not remove) the object at the front (or back) of the deque.

e.g., `x = aDeque.peekFirst()` will do this →



size(), **isEmpty()** and **clear()**

Does the same as with queues

Example:

Consider simulating an "undo list". That is ... many applications allow the user to perform various operations and then select **undo** to undo a mistake that was made. Often, programs will allow you to select undo many times to undo many previous operations in sequence. However, most programs have a limit of some time (e.g., at most 20 undo operations). After the limit is reached, it is impossible to undo the older operations.

In JAVA, we can use an **ArrayDeque** (in the **java.util** package) to maintain our undo deque. Each time the user performs an operation, it is added to the end of the deque. Hence the end of the deque has the most recent operation performed while the front of the deque has the oldest operation performed.

Assuming that we set a limit of perhaps 5 undo operations, the undo deque can hold at most 5 operations. What if a 6th operation is performed ... what do we do? We need to add it to the end of the deque, but then remove the oldest undo operation from the front of the deque.

Here is an example of how we could do this. The example here shows how simple operations (in the form of Strings) can be maintained in an **ArrayDeque** ADT:

```
import java.util.ArrayDeque;

public class DequeTestProgram {
    private static int UNDO_LIST_CAPACITY = 5;

    private static ArrayDeque<String> operations;

    private static void performOperation(String x) {
        if (operations.size() == UNDO_LIST_CAPACITY)
            operations.removeFirst();
        operations.addLast(x);
    }

    private static void undo() {
        operations.removeLast();
    }

    public static void main(String[] args) {
        operations = new ArrayDeque<String>();
        System.out.println("Simulating some cut/paste/move operations ... ");
        performOperation("cut1");
        performOperation("paste1");
        performOperation("move1");
        System.out.println("Here is the undo list now: " + operations);

        System.out.println("Simulating an undo operation ...");
        undo();
        System.out.println("Here is the undo list now: " + operations);

        System.out.println("Simulating another undo operation ...");
        undo();
        System.out.println("Here is the undo list now: " + operations);

        System.out.println("Simulating some more cut/paste/move operations ... ");
        performOperation("cut2");
```

```

performOperation("paste2");
performOperation("move2");
performOperation("move3");
System.out.println("Here is the undo list now: " + operations);

System.out.println("Simulating some more paste operations ... ");
performOperation("paste3");
performOperation("paste4");

System.out.println("Here is the undo list now: " + operations);
}
}

```

Here is the output ... as you can see, the deque is maintained with a size of at most 5:

```

Simulating some cut/paste/move operations ...
Here is the undo list now: [cut1, paste1, move1]
Simulating an undo operation ...
Here is the undo list now: [cut1, paste1]
Simulating another undo operation ...
Here is the undo list now: [cut1]
Simulating some more cut/paste/move operations ...
Here is the undo list now: [cut1, cut2, paste2, move2, move3]
Simulating some more paste operations ...
Here is the undo list now: [paste2, move2, move3, paste3, paste4]

```

8.5 The Stack ADT

Consider the **Stack** ADT:

*A **stack** is an abstract data type that stores elements in a last-in-first-out (LIFO) order. Elements are added and removed to/from the top only.*

A stack stores items one on top of another. When a new item comes in, we place it on the top of the stack and when we want to remove an item, we take the top one from the stack. Stacks are used for many applications in computer science such as syntax parsing, memory management, reversing data, backtracking, etc..

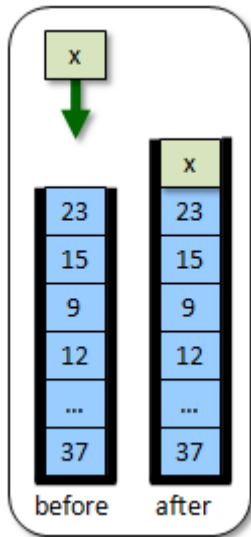
Typical methods for **Stacks** are:



push(Object x)

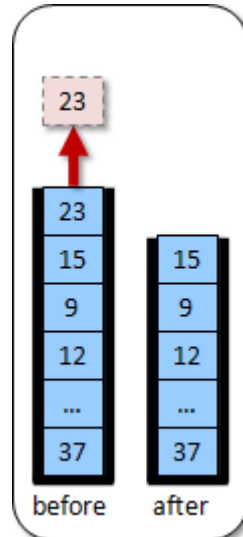
Insert object **x** at the top of the stack.

e.g., `aStack.push(x)`

**pop()**

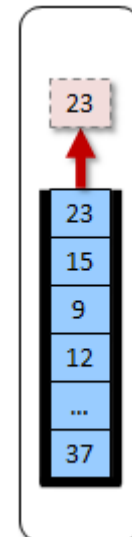
Remove and return the object at the top of the stack.

e.g., `x = aStack.pop()`

**peek()**

Return (but do not remove) the object at the top of the stack.

e.g., `x = aStack.peek()`



`size()`, `empty()` and `clear()`

Does the same as lists

(notice **empty()** instead of **isEmpty()**)

Example:

Consider a math expression that contains numbers, operators and parentheses (i.e., round brackets). How could we write a program that takes a **String** representing a math expression and then determines whether or not the brackets match properly (i.e., each opening bracket has a matching closing bracket in the right order) ?



```

"((23 + 4 * 5) - 34) + (34 - 5)" // no match
"((23 + 4 * 5) - 34) + ((34 - 5)" // no match
"((23 + 4 * 5) - 34) + (34 - 5)" // match

```

How would we approach solving this problem? Well, we need to understand the process. I'm sure that you realize that we need to look at all the String's characters. Perhaps from start to end with a loop, but then what do we do ?

Lets assume that we are not interested in determining whether the formula makes sense but rather that each opening bracket is matched by a closing bracket. Therefore, we are interested in the bracket characters (and), but not the other characters. When encountering

an open bracket as we go through the characters of the string, we need to do something. We might think right away of trying to find the matching closing bracket for each open bracket, but that is not as easy as it sounds. There are many special cases that can be tricky.

A simpler approach would be to make sure that whenever we find a closing bracket, we just need to make sure that we already encountered an open bracket to match with it. This can be done by keeping a count of the number of open brackets. When encountering an opening bracket we increment the counter and when encountering a closing bracket we decrement the counter. If, when all done, the counter is not zero, there is no match. Otherwise the brackets match. Consider these cases:

```
"()" // counter = 0, match
")(" // counter = 1, no match
"(((" // counter = 3, no match
"((())())" // counter = 0, match
"(()))" // counter = -1, no match
"" // counter = 0, match
```

There is a special case that we did not consider. If the counter ever becomes negative before we are done, then we must have encountered a closing bracket before an open bracket ... and there is no match:

```
")(" // counter = -1, no match
"())(" // counter = -1, no match
```

So, how do we write the code? We can use a **FOR** loop and some **IF** statements to check for brackets as follows:

```
public static boolean bracketsMatch(String s) {
    int count = 0;
    char c;

    for (int i=0; i<s.length(); i++) {
        c = s.charAt(i);
        if (c == '(') count++;
        if (c == ')') count--;
        if (count < 0)
            return false;
    }
    return count == 0;
}
```

Here is a test program to try it out:

```

import java.util.*;

public class BracketMatchTestProgram {

    public static boolean bracketsMatch(String s) { /* code as above */ }

    public static void main(String[] args) {
        Scanner    keyboard = new Scanner(System.in);
        String     aString;

        do {
            System.out.println("Please enter expression: (<cr> to quit)");
            aString = keyboard.nextLine();

            if (bracketsMatch(aString))
                System.out.println("The brackets match");
            else
                System.out.println("The brackets do not match");
        } while (aString.length() > 0);
    }
}

```

Here are some testing results:

```

Please enter expression: (<cr> to quit)
((23 + 4 * 5) - 34) + (34 - 5)
The brackets do not match
Please enter expression: (<cr> to quit)
((23 + 4 * 5) - 34) + ((34 - 5)
The brackets do not match
Please enter expression: (<cr> to quit)
((23 + 4 * 5) - 34) + (34 - 5)
The brackets match
Please enter expression: (<cr> to quit)
()
The brackets match
Please enter expression: (<cr> to quit)
()(
The brackets do not match
Please enter expression: (<cr> to quit)
(((
The brackets do not match
Please enter expression: (<cr> to quit)
((())())
The brackets match
Please enter expression: (<cr> to quit)
(())
The brackets do not match
Please enter expression: (<cr> to quit)
)()
The brackets do not match
Please enter expression: (<cr> to quit)
The brackets match

```

I think that our solution works fine. The bracket-matching example above is not very difficult, but what if we have 3 kinds of brackets (, [, and { ? Consider this example:

```
"{2 +(3 *[4 - 5])}" // not supposed to match
```

Maybe we can keep 3 counters ? If we just keep three counters separately, we cannot tell whether the brackets are well formed with respect to one another. We somehow need to know the ordering of each type of bracket so that we can ensure the reverse ordering when we find the closing brackets.

The need for backtracking may seem a little clearer if we consider a different application of the bracket matching program. Suppose that we want to match the brackets in our JAVA code...

```
public class PrintWriterTestProgram {
    public static void main (String[] args) {
        try {
            BankAccount aBankAccount;
            PrintWriter out;

            aBankAccount = new BankAccount ("Rob Banks");
            aBankAccount.deposit (100);
            out = new PrintWriter (new FileWriter ("myAccount2.dat"));
            out.println (aBankAccount.getOwner ());
            out.println (aBankAccount.getAccountNumber ());
            out.println (aBankAccount.getBalance ());
            out.close ();
        } catch (FileNotFoundException e) {
            System.out.println ("Error: Cannot open file for writing");
        } catch (IOException e) {
            System.out.println ("Error: Cannot write to file");
        }
    }
}
```

Here we see, for example, that the portion of code inside the **class** definition must have all of its brackets matching, and that involves matching the code inside the **main** method's body and then within the **try** block etc... The compiler does this kind of bracket matching to make sure that your code is well-formed.

The **stack** data structure is designed for this purpose. It allows us to back-track ... which is essentially what we need to do when finding a closing bracket. Here is how we can use a stack. When we find an open bracket, we put it on the top of the stack, regardless of its type. When we find a closing bracket, we take the top opening bracket from the stack and check to see if it is the same type as the closing bracket. If not, the brackets are in the wrong order, otherwise all is fine and we continue onwards. If, when encountering a closing bracket, we find that the stack is empty, then there is no match either.

Lets look at the code. In JAVA, we make a **Stack** by simply calling its constructor:

```
Stack aStack = new Stack();
```

However, in our case, we are going to be placing bracket character on the Sstack. Therefore we should specify this as follows:

```
Stack<Character> aStack = new Stack<Character>();
```

Then, we need to use the appropriate **Stack** methods: Here is the resulting code:

```
public static boolean bracketsMatch2(String s) {
    Stack<Character> aStack;
    char c;

    aStack = new Stack<Character>();
    for (int i=0; i<s.length(); i++) {
        c = s.charAt(i);

        if ((c == '(') || (c == '[') || (c == '{')) // got open bracket
            aStack.push(c);

        if ((c == ')') || (c == ']') || (c == '}')) { // got closed bracket
            if (aStack.empty())
                return false; // no open bracket for this closed one

            char top = aStack.pop(); // get the last opening bracket found
            if ((c == ')') && (top != '(') ||
                (c == ']') && (top != '[') ||
                (c == '}') && (top != '{'))
                return false; // wrong closing bracket for last opened one
        }
    }
    return aStack.empty(); // No match if brackets are left over
}
```

Notice in the above code that it never has **return true** anywhere. In fact, it is only at the very end, once we have checked all characters that there is a chance for the method to return **true**. This will happen if the stack is empty (i.e., all open brackets have been matched with closing ones). If desired, you can simplify the above code by replacing the **IF** statements with a **SWITCH** statement as follows:

```
switch(c) {
    case '(':
    case '[':
    case '{':
        aStack.push(c); // got open bracket
        break;
    case ')':
        if (aStack.empty() || (aStack.pop() != '('))
            return false;
        break;
    case ']':
```

```

    if (aStack.empty() || (aStack.pop() != '['))
        return false;
    break;
case '}':
    if (aStack.empty() || (aStack.pop() != '{'))
        return false;
    break;
}

```

Here are some testing results that we would obtain if we replaced our previous `bracketsMatch()` method with this new method:

```

Please enter the expression: (just <cr> to quit)
](){[
The brackets do not match
Please enter the expression: (just <cr> to quit)
()[]{
The brackets match
Please enter the expression: (just <cr> to quit)
{((([[]])))}
The brackets match
Please enter the expression: (just <cr> to quit)
{[[]][
The brackets do not match
Please enter the expression: (just <cr> to quit)
}]]]}
The brackets do not match
Please enter the expression: (just <cr> to quit)
((() [{}]) [() [{}])
The brackets do not match
Please enter the expression: (just <cr> to quit)
The brackets match

```

Challenge: Could you adjust the code above to read in a JAVA file instead of using a fixed string and have it ensure that the brackets match ?

8.6 The Set ADT

Consider the **Set** ADT:

*A **set** is an abstract data type that does not allow duplicate elements to be added.*

That is, there cannot be two elements e_1 and e_2 such that $e_1.equals(e_2)$. Any attempt to add duplicate elements is ignored. Sets differ from Lists in that the elements are not kept in the same order as when they were added. Sets are generally unordered, which means that the particular location of an element may change according to the particular set implementation.



Typical operations for **Sets** are:

- `add(Object x)`
- `remove(Object x)`

These operations work similar to that of **Lists** with the exception that the elements are not necessarily maintained in the same order that they were added in. Also, items are not guaranteed to be added to the **Set** because the `add()` operation will not allow any duplicate items to be added.

Example:

Suppose we wanted to maintain a list of DVD titles in our personal movie collection. Likely we do not want to have two or more of the same DVD. We can use a **Set** to avoid duplicates. In JAVA, we use the **HashSet** class which is in `java.util` package. Consider a simple **Movie** class as follows:

```
public class Movie {
    private String    title;

    public Movie(String t) { title = t; }
    public String getTitle() { return title; }

    public String toString() {
        return "Movie: \"" + title + "\"";
    }
}
```

Now consider the following code which simulates some inventory at a video store. The code makes use of the simple **Movie** object by adding **10** movies from among **5** unique titles ... hence many duplicates. The code makes use of `Math.random()` so that the inventory is different each time we run the program.

```
import java.util.*;
public class SetTestProgram1 {
    public static void main(String[] args) {
        Movie[] dvds = {new Movie("Bolt"),
                        new Movie("Monsters Vs. Aliens"),
                        new Movie("Marley & Me"),
                        new Movie("Hotel For Dogs"),
                        new Movie("The Day the Earth Stood Still")};
        ArrayList<Movie> inventory = new ArrayList<Movie>();

        // Add 10 random movies from the list of dvds
        for (int i=0; i<10; i++) {
            inventory.add(dvds[(int) (Math.random()*5)]);
        }

        for (Movie m: inventory)
            System.out.println(m);
    }
}
```

Of course, each time that we run this code the result is different. Here is an example of what we may see:

```
Movie: "Bolt"
Movie: "The Day the Earth Stood Still"
Movie: "Bolt"
Movie: "Marley & Me"
Movie: "Monsters Vs. Aliens"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "Bolt"
Movie: "Bolt"
Movie: "Bolt"
```

Can we adjust the **for** loop so that it only displays unique movies? No. We would have to do some extra work of making a new list with the duplicates removed. So, we could replace the **for** loop with the following:

```
ArrayList<Movie> uniqueList = new ArrayList<Movie>();

for (Movie m: inventory) {
    if (!uniqueList.contains(m))
        uniqueList.add(m);
}
for (Movie m: uniqueList) {
    System.out.println(m);
}
```

This would produce the following output (according to the earlier results):

```
Movie: "Bolt"
Movie: "The Day the Earth Stood Still"
Movie: "Marley & Me"
Movie: "Monsters Vs. Aliens"
```

However, there is an easier way to do this. Consider what happens if we change the inventory from an **ArrayList** to a **HashSet** as follows:

```
HashSet<Movie> inventory = new HashSet<Movie>();
```

Our code would produce the following output (which varies according to the randomness):

```
Movie: "Monsters Vs. Aliens"
Movie: "Bolt"
Movie: "The Day the Earth Stood Still"
Movie: "Marley & Me"
```

Notice that the duplicates were removed. The **HashSet** prevented any duplicates from being added. Therefore, we have lost all duplicate copies from our inventory, which can be bad.

Perhaps it would be better to only use a **HashSet** when displaying the inventory, so that we don't destroy the duplicate movies. This is easily done by creating an extra **HashSet** variable (**displayList** in this case) and using the **HashSet** constructor that takes a **Collection** parameter:

```
import java.util.*;

public class SetTestProgram2 {
    public static void main(String[] args) {
        Movie[] dvds = {new Movie("Bolt"),
                        new Movie("Monsters Vs. Aliens"),
                        new Movie("Marley & Me"),
                        new Movie("Hotel For Dogs"),
                        new Movie("The Day the Earth Stood Still")};

        ArrayList<Movie> inventory = new ArrayList<Movie>();

        // Add 10 random movies from the list of dvds
        for (int i=0; i<10; i++) {
            inventory.add(dvds[(int) (Math.random()*5)]);
        }

        System.out.println("Here are the unique movies:");
        HashSet<Movie> displayList = new HashSet<Movie>(inventory);
        for (Movie m: displayList)
            System.out.println(m);

        System.out.println("\nHere is the whole inventory:");
        for (Movie m: inventory)
            System.out.println(m);
    }
}
```

Notice the parameter to the **HashSet** constructor. This constructor will ensure to add all the elements from the inventory collection to the newly create **HashSet**. Then, in the **FOR** loop, we use this new **HashSet** for display purposes, while the original inventory remains unaltered. Here is the output:

```
Here are the unique movies:
Movie: "The Day the Earth Stood Still"
Movie: "Bolt"
Movie: "Monsters Vs. Aliens"
Movie: "Hotel For Dogs"

Here is the whole inventory:
Movie: "Bolt"
Movie: "Bolt"
Movie: "Bolt"
Movie: "Monsters Vs. Aliens"
Movie: "The Day the Earth Stood Still"
Movie: "Hotel For Dogs"
Movie: "Bolt"
Movie: "Monsters Vs. Aliens"
Movie: "Monsters Vs. Aliens"
Movie: "Bolt"
```

So, it is easy to remove duplicates from any collection ... we simply create a new **HashSet** from the collection and it removes the duplicates for us.

However, there is one point that should be mentioned. In the above code, the duplicate movies all represented the same exact object in memory ... that is ... all duplicates were identical to one another. However, it is more common to have two equal movies which are not identical. So, consider this code ... notice the equal (but not identical) **Movie** objects:

```
import java.util.*;

public class SetTestProgram3 {
    public static void main(String[] args) {
        Movie[] dvds = {new Movie("Bolt"),
                        new Movie("Monsters Vs. Aliens"),
                        new Movie("Marley & Me"),
                        new Movie("Monsters Vs. Aliens"),
                        new Movie("Hotel For Dogs"),
                        new Movie("Hotel For Dogs"),
                        new Movie("Monsters Vs. Aliens"),
                        new Movie("The Day the Earth Stood Still"),
                        new Movie("The Day the Earth Stood Still"),
                        new Movie("The Day the Earth Stood Still"),
                        new Movie("The Day the Earth Stood Still")};

        ArrayList<Movie> inventory = new ArrayList<Movie>();

        // Add 10 random movies from the list of dvds
        for (int i=0; i<10; i++) {
            inventory.add(dvds[(int) (Math.random()*11)]);
        }

        System.out.println("Here are the unique movies:");
        HashSet<Movie> displayList = new HashSet<Movie>(inventory);
        for (Movie m: displayList)
            System.out.println(m);

        System.out.println("\nHere is the whole inventory:");
        for (Movie m: inventory)
            System.out.println(m);
    }
}
```

Here is the result:

```
Here are the unique movies:
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "Bolt"
Movie: "Marley & Me"
Movie: "Monsters Vs. Aliens"
Movie: "Monsters Vs. Aliens"
```

```
Here is the whole inventory:
Movie: "Marley & Me"
Movie: "Monsters Vs. Aliens"
Movie: "The Day the Earth Stood Still"
```

```

Movie: "Monsters Vs. Aliens"
Movie: "Bolt"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"

```

Notice that there are many duplicates still in the **Set**. The problem is that the **Set** classes make use of a particular method in order to determine whether or not an object is the same as another one already in the set. The method used by java to determine equality between objects is called the **equals(Object x)** method. So, in general, the **equals()** method can be used to compare any two objects to determine whether or not they are the same:

```

anObject.equals(anotherObject)    // returns either true or false

```

All objects in JAVA have a default inherited **equals()** method which resides in the **Object** class. Unfortunately however, this default **equals()** method does the following:

```

public boolean equals(Object x) {
    return (this == x);
}

```

Therefore, the method, by default, will only return **true** if the object is the same exact object as **x** (i.e., the same memory location reference). Therefore, when we compare two **Movie** objects that were created using their own constructors, these movies can never be equal by default since they each reside in their own individual memory locations.

```

Movie m1 = new Movie("Monsters Vs. Aliens");
Movie m2 = new Movie("Monsters Vs. Aliens");
Movie m3 = m2;
m1.equals(m2);    // returns false
m2.equals(m3);    // returns true

```

In order to avoid this problem, we need to re-define the **equals()** method for our example. That is, we need to make our own **equals()** method for the **Movie** objects that overrides the one up in the **Object** class. To do this, we simply write the following method in the **Movie** class:

```

public boolean equals(Object x) {
    if (!(x instanceof Movie))
        return false;
    return title.equals(((Movie)x).title);
}

```

Notice that the method returns a **boolean** and takes a single parameter of type **Object**. This is a general parameter that allows us to compare the **Movie** with any type of object. Of course, if the parameter is not actually a **Movie** object, then the result should be **false**. This test is done using the **instanceof** keyword in JAVA that determines whether an object is an instance of a particular class.

Once we do that, we then need to decide what it means for two movies to be considered equal. For simplicity, we will assume that two **Movie** objects are equal if they have the same title. Notice that we simply ask the movies for their titles and then use the **equals()** method from the **String** class (already written to compare characters). One added point is that we need to *typecast* the parameter **x** to a **Movie** object.

Logically, the addition of this **equals()** method should solve the problem. However, it does not quite. As it turns out, in JAVA, **Sets** make use of a programming technique called **hashing**. Hashing is used as a way of quickly comparing and sorting objects because it quickly identifies objects that cannot be equal to one another, without needing to go deep down inside the object to make comparisons. For example, if you had an apple and a pineapple, they are clearly not equal. You need not compare them closely because a simple quick glance tells you that they are not the same.



In real life, hashing is used by post offices when sorting mail at various levels. First, they look at the destination country and make two piles ... domestic mail vs. international mail. That is a quick “hash” in that the postmen do not need to examine any further details at that time ... such as street names and recipient names etc... Then they hash again later by using the *postal code* to determine “roughly” and “quickly” the area of a city that your mail needs to be delivered to. This allows them to make a pile of mail for all people living in the same area. At each level of “sorting” the mail (i.e., country, city, postal code, street), the postmen must make a quick decision as to which pile to place the mail item into. This quick decision is based on something called a **hash function** (or **hash code**).



In JAVA, for **Sets** to work properly, we must also write a **hashCode()** method for our objects. These methods return an **int** which represents the “pile” that the object belongs to. Similar objects will have similar hash codes, and therefore end up in the same “pile”. Here is the **hashCode()** method for our **Movie** object:

```
public int hashCode() {
    return title.hashCode();
}
```

It must be **public**, return an **int** and have no parameters. The code simply returns the hash code of the title string for the movie. We do not wish to go into details here as to “how” to produce a proper hash code. Instead, let us simply use this “rule of thumb”: the hash code for our objects should return a sum of all the hash codes of its attributes. If an attribute is a primitive, just convert it to an integer in some way and use that value in the **hashCode()** method’s total value.

Now, the code in **SetTestProgram3** will work as desired ... removing all duplicates.

Example:

You will notice in the previous example, that the **HashSet** did not sort the items. Also, the items don't even appear in the order that they were added. Instead, the order seems somewhat random and arbitrary.

If you want the items in sorted order, you can use a **TreeSet** instead of a **HashSet**:

```
TreeSet<Movie> displayList = new TreeSet<Movie>(inventory);
```

Of course, as we did with **PriorityQueues**, we will need to make sure that our **Movie** class implements the **Comparable<Movie>** interface and thus has a **compareTo()** method. Here is the completed **Movie** class that will work with both **HashSet** and **TreeSet**:

```
public class Movie implements Comparable<Movie> {
    private String title;

    public Movie(String t) { title = t; }
    public String getTitle() { return title; }
    public String toString() { return "Movie: \"" + title + "\""; }

    public boolean equals(Object obj) {
        if (!(obj instanceof Movie)) return false;
        return title.equals(((Movie)obj).title);
    }

    public int hashCode() {
        return title.hashCode();
    }

    public int compareTo(Movie m) {
        return title.compareTo(m.title);
    }
}
```

Here is the output from our **SetTestProgram3** when using **TreeSet** instead of **HashSet**:

Here are the unique movies:

```
Movie: "Bolt"
Movie: "Hotel For Dogs"
Movie: "Marley & Me"
Movie: "Monsters Vs. Aliens"
Movie: "The Day the Earth Stood Still"
```

Here is the whole inventory:

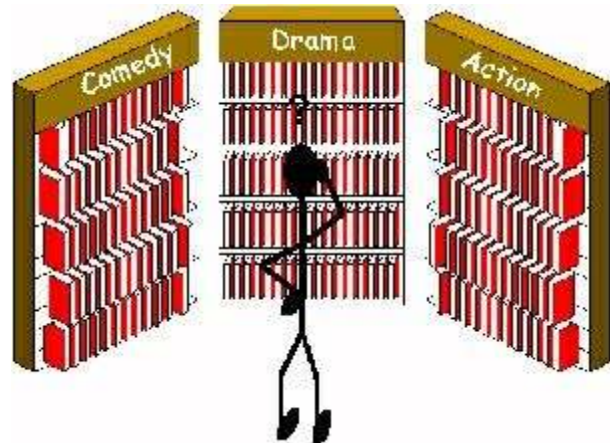
```
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "Marley & Me"
Movie: "Monsters Vs. Aliens"
Movie: "Hotel For Dogs"
```

Movie: "Monsters Vs. Aliens"
 Movie: "Marley & Me"
 Movie: "Bolt"

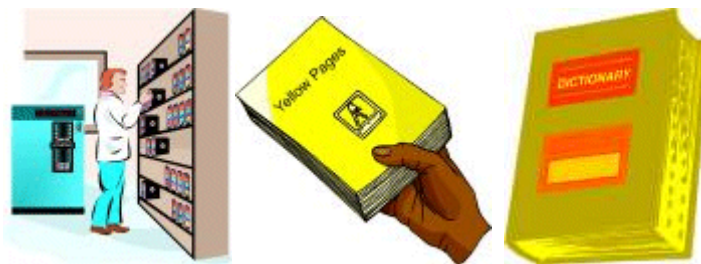
Notice the sorted order of the movies from the **TreeSet**.

8.7 The Dictionary/Map ADT

Sometimes we want to store data in an organized manner that allows us to quickly find what we are looking for. For example, consider a video store. Isn't it a nice idea to have the movies arranged by category so that you don't waste time looking over movies that are not of an interesting nature (such as musicals or perhaps drama) ?



You may also agree that it is easy to find a definition of a word in a dictionary because the definition is paired up with the word that it defines. Once we know the word that we want to look up, then we can find its definition. The word is therefore the unique **key** to finding the definition. We say that the definition of the word is the **value** associated with that key word. Likewise, a person's phone number is paired up with his/her name in a phonebook so that we can find numbers easily based the person's name as the key identifier. We say that the phone number is the value for the particular key person.



This idea of a key-value pairing (or mapping) is the basis of the **Dictionary** ADT:

*A **dictionary** is an abstract data type that stores a collection of unique keys and their associated values. Each **key** is associated with a single **value** or a set of values.*

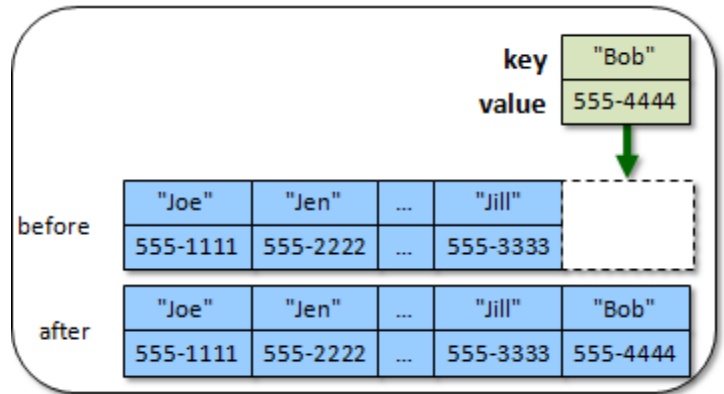
The key is always necessary in order to access a particular value in the dictionary. Like the **Collection** interface, the **Map** interface stores objects as well. So what is different? Well, a **Map** stores things in a particular way such that the objects can be easily located later on. A **Map** really means a "Mapping" of one object to another. A **Map** does not keep items in any particular index order, as with lists. In a way, however, instead of having integer indices, a **Map** has arbitrary objects as indices. So just as a unique index into a list or array identifies the object at that location, the **key** in the **Map** identifies the object associated with it.

The basic methods for inserting, removing, accessing and modifying items from a **Map** are as follows:

put(Object k, Object v)

Place object **v** in the map with key **k**. If there is already a value at key **k**, it is replaced by **v**.

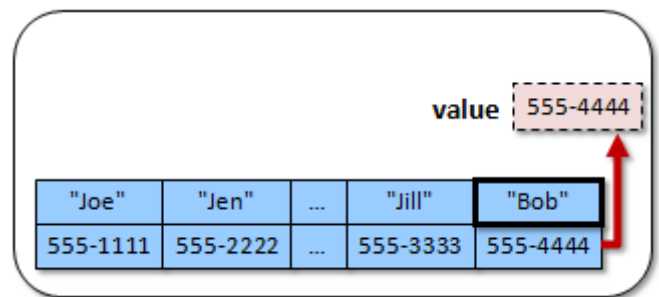
e.g., `aMap.put("Bob", "555-4444")` will do this →



get(Object k)

Return the value currently associated with key **k**. If **k** is not in the map yet, usually **null** is returned.

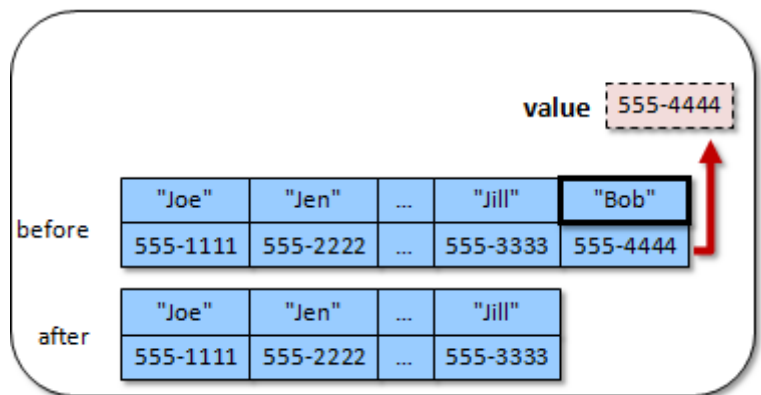
e.g., `v = aMap.get("Bob")` will return **v** →



remove(Object k)

Remove the key/value pair associated with key **k** from the Map. Usually, the value associated with the key is returned.

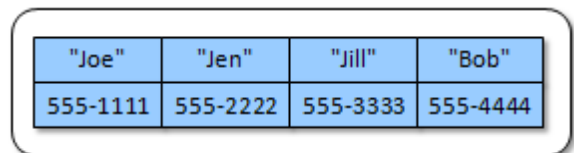
e.g., `v = aMap.remove("Bob")` will do this →



size()

Return the number of keys in the list.

e.g., `n = aMap.size()` will return **4** →



clear()

Remove all elements from the list.

There are additional methods often available for convenience sake. Here are some:

containsKey(Object k)

Return **true** if there is an entry in the map with key **k**, otherwise return **false**.

e.g., `b = aMap.containsKey("Bob")`
will return **true** →
e.g., `b = aMap.containsKey("Max")`
will return **false** →

"Joe"	"Jen"	"Jill"	"Bob"
555-1111	555-2222	555-3333	555-4444

containsValue(Object v)

Return **true** if there is an entry in the map with value **v**, otherwise return **false**.

e.g., `b = aMap.containsValue("555-1111")`
will return **true** →
e.g., `b = aMap.containsValue("Jill")`
will return **false** →

"Joe"	"Jen"	"Jill"	"Bob"
555-1111	555-2222	555-3333	555-4444

keySet()

Return a **Set** containing all keys in the map.

e.g., `s = aMap.keySet()`
will return ["Joe", "Jen", "Jill", "Bob"] →

"Joe"	"Jen"	"Jill"	"Bob"
555-1111	555-2222	555-3333	555-4444

values()

Return a **Collection** of all values in the map.

e.g., `c = aMap.values()`
will return ["555-1111", "555-2222",
"555-3333", "555-4444"] →

"Joe"	"Jen"	"Jill"	"Bob"
555-1111	555-2222	555-3333	555-4444

isEmpty()

Return **true** if the number of elements in the list is **0**, otherwise return **false**.

does the same as this:

```
return (aMap.size() == 0);
```

In JAVA, the *Map* ADT is called a **HashMap** and it is located in the **java.util** package, which must be imported in order to use this data type. To create a **HashMap**, we can simply call a constructor from the **HashMap** class. Here is an example of creating a **HashMap** and storing it in a variable so that we can use it:

```
HashMap myMap;
myMap = new HashMap();
```

However, we usually indicate the type of the key and the value when we declare the map:

```
HashMap<String, Integer> myMap;
myMap = new HashMap<String, Integer>();
```

There is also a **TreeMap** class that represents a sorted Map. That is, it maintains the keys in sorted order.

Example:

Consider a program that keeps track of some people and their favorite movie. We can associate a **Movie** object with each **Person** object and store them in a **HashMap** as follows:

```
import java.util.*;
public class MapTestProgram {
    public static void main(String args[]) {
        HashMap<Person, Movie> favMovies = new HashMap<Person, Movie>();
        Person pete, jack;

        favMovies.put(pete = new Person("Pete Zaria", 12),
            new Movie("Monsters Vs. Aliens"));
        favMovies.put(new Person("Rita Book", 20),
            new Movie("Marley & Me"));
        favMovies.put(new Person("Willie Maykit", 65),
            new Movie("Monsters Vs. Aliens"));
        favMovies.put(new Person("Patty O'Furniture", 41),
            new Movie("Hotel For Dogs"));
        favMovies.put(new Person("Sue Permann", 73),
            new Movie("Monsters Vs. Aliens"));
        favMovies.put(new Person("Sid Down", 19),
            new Movie("Bolt"));
        favMovies.put(jack = new Person("Jack Pot", 4),
            new Movie("Bolt"));

        System.out.println("There are: " + favMovies.size() + " favorite movies");
        System.out.println("Pete's favorite movie is: " + favMovies.get(pete));
        System.out.println("Jack's favorite movie is: " + favMovies.get(jack));
        System.out.println("The Map keys are:" + favMovies.keySet());
        System.out.println("The Map values are:" + favMovies.values());
        System.out.println("Removing Pete from the list ...");
        favMovies.remove(pete);
        System.out.println("Pete's favorite movie is: " + favMovies.get(pete));
        System.out.println("Is Pete in the Map ? " + favMovies.containsKey(pete));
        System.out.print("Is anyone's favorite Star Trek ? ");
        System.out.println(favMovies.containsValue(new Movie("Star Trek")));
        System.out.print("Is anyone's favorite Bolt ? ");
        System.out.println(favMovies.containsValue(new Movie("Bolt")));
    }
}
```

Here is the output:

```

There are: 7 favorite movies
Pete's favorite movie is: Movie: "Monsters Vs. Aliens"
Jack's favorite movie is: Movie: "Bolt"
The Map keys are:[4 year old Jack Pot, 65 year old Willie Maykit, 20 year old
Rita Book, 19 year old Sid Down, 73 year old Sue Permann, 41 year old Patty
O'Furniture, 12 year old Pete Zaria]
The Map values are:[Movie: "Bolt", Movie: "Monsters Vs. Aliens", Movie:
"Marley & Me", Movie: "Bolt", Movie: "Monsters Vs. Aliens", Movie: "Hotel For
Dogs", Movie: "Monsters Vs. Aliens"]
Removing Pete from the list ...
Pete's favorite movie is: null
Is Pete in the Map ? false
Is anyone's favorite Star Trek ? false
Is anyone's favorite Bolt ? true

```

Example:

Consider an application which represents a movie store that maintains movies to be rented out. Assume that we have a collection of movies. When renting, we would like to be able to find movies quickly. For example, we may want to:



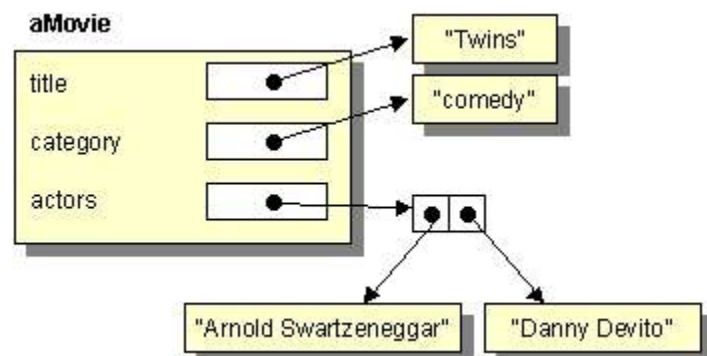
- ask for a movie by title and have it found right away
- search for movies in a certain category (e.g., new release, comedy, action)
- find movies containing a specific actor/actress (e.g., Jackie Chan, Peter Sellers etc...)

Obviously, we could simply store all moves in one big **ArrayList**. But how much time would we waste finding our movies? Imagine a video store in which the movies are not sorted in any particular order ... just randomly placed on the shelves !! We would have to go through them one by one !!!



We will use **HashMaps** to store our movies efficiently so that we can quickly get access to the movies that we want.

Let us start out with the representation of a **Movie** object. Each movie will maintain a **title**, list of **actors** and a **type** (category). Obviously, in a real system, we would need to keep much more information such as ID, rental history, new releases vs. oldies, etc... Here is the diagram representing the **Movie** object:



Let us now define this **Movie** class.

```

import java.util.*;
public class Movie {
    private String          title, type;
    private ArrayList<String> actors;

    public String getTitle() { return title; }
    public String getType() { return type; }
    public ArrayList<String> getActors() { return actors; }

    public Movie() { this("???", "???"); }

    public Movie(String aTitle, String aType) {
        title = aTitle;
        type = aType;
        actors = new ArrayList<String>();
    }

    public String toString() { return "Movie: \"" + title + "\""; }
    public void addActor(String anActor) { actors.add(anActor); }
}

```

Now lets look at the **addActor()** method. It merely adds the given actor (just a name) to the actors arrayList. We can make some example methods to represent some movies. Add the following methods to the **Movie** class:

```

public static Movie example1() {
    Movie aMovie = new Movie("The Matrix", "SciFic");
    aMovie.addActor("Keanu Reeves");
    aMovie.addActor("Laurence Fishburne");
    aMovie.addActor("Carrie-Anne Moss");
    return aMovie;
}

public static Movie example2() {
    Movie aMovie = new Movie("Blazing Saddles", "Comedy");
    aMovie.addActor("Cleavon Little");
    aMovie.addActor("Gene Wilder");
    return aMovie;
}

public static Movie example3() {
    Movie aMovie = new Movie("The Matrix Reloaded", "SciFic");
    aMovie.addActor("Keanu Reeves");
    aMovie.addActor("Laurence Fishburne");
    aMovie.addActor("Carrie-Anne Moss");
    return aMovie;
}

public static Movie example4() {
    Movie aMovie = new Movie("The Adventure of Sherlock Holmes' Smarter
Brother", "Comedy");
    aMovie.addActor("Gene Wilder");
    aMovie.addActor("Madelaine Kahn");
    aMovie.addActor("Marty Feldman");
    aMovie.addActor("Dom DeLuise");
    return aMovie;
}

```

```
public static Movie example5() {
    Movie aMovie = new Movie("The Matrix Revolutions","SciFic");
    aMovie.addActor("Keanu Reeves");
    aMovie.addActor("Laurence Fishburne");
    aMovie.addActor("Carrie-Anne Moss");
    return aMovie;
}

public static Movie example6() {
    Movie aMovie = new Movie("Meet the Fockers","Comedy");
    aMovie.addActor("Robert De Niro");
    aMovie.addActor("Ben Stiller");
    aMovie.addActor("Dustin Hoffman");
    return aMovie;
}

public static Movie example7() {
    Movie aMovie = new Movie("Runaway Jury","Drama");
    aMovie.addActor("John Cusack");
    aMovie.addActor("Gene Hackman");
    aMovie.addActor("Dustin Hoffman");
    return aMovie;
}

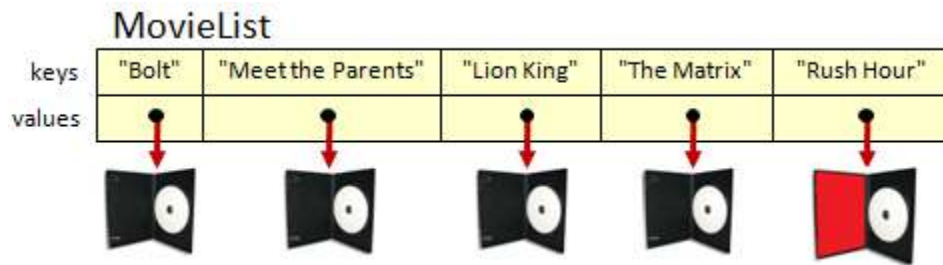
public static Movie example8() {
    Movie aMovie = new Movie("Meet the Parents","Comedy");
    aMovie.addActor("Robert De Niro");
    aMovie.addActor("Ben Stiller");
    aMovie.addActor("Teri Polo");
    aMovie.addActor("Blythe Danner");
    return aMovie;
}

public static Movie example9() {
    Movie aMovie = new Movie("The Aviator","Drama");
    aMovie.addActor("Leonardo DiCaprio");
    aMovie.addActor("Cate Blanchett");
    return aMovie;
}

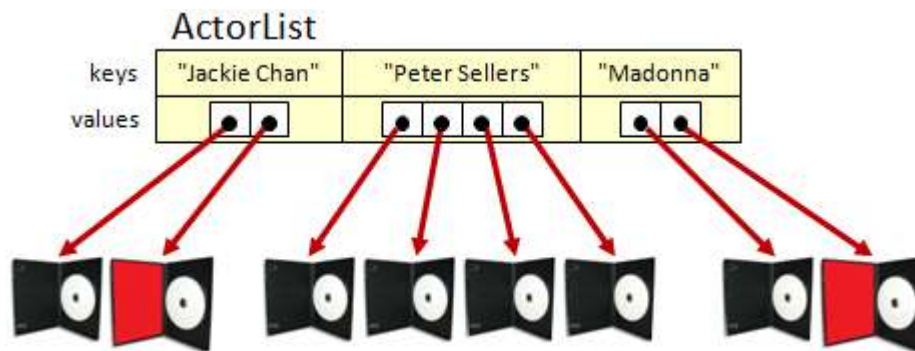
public static Movie example10() {
    Movie aMovie = new Movie("Envy","Comedy");
    aMovie.addActor("Ben Stiller");
    aMovie.addActor("Jack Black");
    aMovie.addActor("Rachel Weisz");
    aMovie.addActor("Amy Poehler");
    return aMovie;
}
```

Now we need to consider the making a **MovieStore** object. Recall, that we want to store movies efficiently using **HashMaps**.

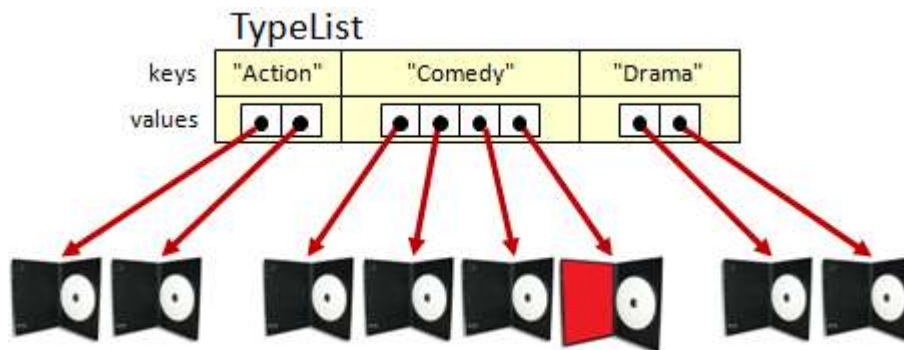
For the **MovieStore**, we will maintain three **HashMaps**. One will be the movieList where the keys are titles and the values are the movie objects with that title.



The second will be the **actorList** which will keep actor/actress names as keys and the values will be ArrayLists of all movies that the actor/actress stars in.



The last one will be the **typeList** in which the keys will be the "types" (or categories) of movies and the values will be ArrayLists of all movies belonging to that type.



Notice that one of the movies is "red" in the picture. Why? This represents the same exact movie. So in fact, the reference to this movie is stored in 4 different places.

Isn't this wasteful? Keep in mind that we are not duplicating all the movie's data ... we are only duplicating the pointer to the movie. So in fact, each time we duplicate a movie in our **HashMaps**, we are simply duplicating its reference (or pointer) which takes 4 bytes.

So, yes, we are taking slightly more space, but at the benefit of allowing quick access to the data. You will learn more about efficiency when you do your second-year course on data structures.

The basic **MovieStore** definition is as follows:

```

import java.util.*;

public class MovieStore {
    private HashMap<String,Movie>          movieList;
    private HashMap<String,ArrayList<Movie>> actorList;
    private HashMap<String,ArrayList<Movie>> typeList;

    public MovieStore() {
        movieList = new HashMap<String,Movie>();
        actorList = new HashMap<String,ArrayList<Movie>>();
        typeList = new HashMap<String,ArrayList<Movie>>();
    }

    public HashMap<String,Movie> getMovieList() { return movieList; }
    public HashMap<String,ArrayList<Movie>> getActorList() { return actorList; }
    public HashMap<String,ArrayList<Movie>> getTypeList() { return typeList; }

    public String toString() {
        return ("MovieStore with " + movieList.size() + " movies.");
    }
}

```

Why do not we need "set" methods for the **HashMaps** ? You should be able to reason on that. Now, how do we add a movie to the store ? Well ... how do the instance variables change ?

- movie must be added to movieList
- movie must be added to typeList. What if it is the first/last movie from this category ?
- movie must be added to actorList. What if it is the first/last movie for this actor ?

Here is the code:

```

//This method adds a movie to the movieStore.
public void addMovie(Movie aMovie) {
    //Add to the movieList
    movieList.put(aMovie.getTitle(), aMovie);

    //If there is no category matching this movie's type, make a new category
    if (!typeList.containsKey(aMovie.getType()))
        typeList.put(aMovie.getType(), new ArrayList<Movie>());

    //Add the movie to the proper category.
    typeList.get(aMovie.getType()).add(aMovie);

    //Now add all of the actors
    for (String anActor: aMovie.getActors()) {
        //If there is no actor yet matching this actor, make a new actor key
        if (!actorList.containsKey(anActor))
            actorList.put(anActor, new ArrayList<Movie>());

        //Add the movie for this actor
        actorList.get(anActor).add(aMovie);
    }
}

```


In fact, removing a movie is just as easy:

```
//This private method removes a movie from the movie store
private void removeMovie(Movie aMovie) {
    //Remove from the movieList
    movieList.remove(aMovie.getTitle());

    //Remove from the type list vector.  If last one, remove the type.
    typeList.get(aMovie.getType()).remove(aMovie);
    if (typeList.get(aMovie.getType()).isEmpty())
        typeList.remove(aMovie.getType());

    //Now Remove from the actors list.  If actor has no more, remove him.
    for(String anActor: aMovie.getActors()) {
        actorList.get(anActor).remove(aMovie);
        if (actorList.get(anActor).isEmpty())
            actorList.remove(anActor);
    }
}
```

However, what if we do not have a hold of the **Movie** object that we want to delete ? Perhaps we just know the title of the movie that needs to be removed. We can write a method which asks to remove a movie with a certain title. All it needs to do is grab a hold of the movie and then call the remove method that we just wrote.

```
//This method removes a movie (given its title) from the movie store
public void removeMovieWithTitle(String aTitle) {
    if (movieList.get(aTitle) == null)
        System.out.println("No movie with that title");
    else
        removeMovie(movieList.get(aTitle));
}
```

Well, perhaps the final thing we need to do is list the movies (or print them out). How do we do this ? What if we want them in some kind of order ? Perhaps any order, by actor/actress, or by type. Here's how to display them in the order that they were added to the **MovieStore**:

```
//This method lists all movie titles that are in the store
public void listMovies() {
    for (String s: movieList.keySet())
        System.out.println(s);
}
```

What about listing movies that star a certain actor/actress ? Well it just requires an additional search. Can you guess which **HashMap** is needed ?

```
//This method lists all movies that star the given actor
public void listMoviesWithActor(String anActor) {
    for (Movie m: actorList.get(anActor))
        System.out.println(m);
}
```

Lastly, let us list all of the movies that belong to a certain category (type). For example, someone may wish to have a list of all comedy movies in the store. It is actually very similar to the actor version.

```
//This method lists all movies that have the given type
public void listMoviesOfType(String aType) {
    for (Movie m: typeList.get(aType))
        System.out.println(m);
}
```

Ok, now we better test everything:

```
public class MovieStoreTester {
    public static void main(String args[]) {
        MovieStore aStore = new MovieStore();
        aStore.addMovie(Movie.example1());
        aStore.addMovie(Movie.example2());
        aStore.addMovie(Movie.example3());
        aStore.addMovie(Movie.example4());
        aStore.addMovie(Movie.example5());
        aStore.addMovie(Movie.example6());
        aStore.addMovie(Movie.example7());
        aStore.addMovie(Movie.example8());
        aStore.addMovie(Movie.example9());
        aStore.addMovie(Movie.example10());

        System.out.println("Here are the movies in: " + aStore);
        aStore.listMovies();
        System.out.println();

        //Try some removing now
        System.out.println("Removing The Matrix");
        aStore.removeMovieWithTitle("The Matrix");
        System.out.println("Trying to remove Mark's Movie");
        aStore.removeMovieWithTitle("Mark's Movie");

        //Do some listing of movies
        System.out.println("\nHere are the Comedy movies in: " + aStore);
        aStore.listMoviesOfType("Comedy");
        System.out.println("\nHere are the Science Fiction movies in: " + aStore);
        aStore.listMoviesOfType("SciFic");
        System.out.println("\nHere are the movies with Ben Stiller:");
        aStore.listMoviesWithActor("Ben Stiller");
        System.out.println("\nHere are the movies with Keanu Reeves:");
        aStore.listMoviesWithActor("Keanu Reeves");
    }
}
```

Here is the output:

```
Here are the movies in: MovieStore with 10 movies.
The Matrix Revolutions
Runaway Jury
The Matrix
The Adventure of Sherlock Holmes' Smarter Brother
The Aviator
```

```
The Matrix Reloaded
Blazing Saddles
Meet the Parents
Envy
Meet the Fockers
```

```
Removing The Matrix
Trying to remove Mark's Movie
No movie with that title
```

```
Here are the Comedy movies in: MovieStore with 9 movies.
Movie: "Blazing Saddles"
Movie: "The Adventure of Sherlock Holmes' Smarter Brother"
Movie: "Meet the Fockers"
Movie: "Meet the Parents"
Movie: "Envy"
```

```
Here are the Science Fiction movies in: MovieStore with 9 movies.
Movie: "The Matrix Reloaded"
Movie: "The Matrix Revolutions"
```

```
Here are the movies with Ben Stiller:
Movie: "Meet the Fockers"
Movie: "Meet the Parents"
Movie: "Envy"
```

```
Here are the movies with Keanu Reeves:
Movie: "The Matrix Reloaded"
Movie: "The Matrix Revolutions"
```

8.8 Collections Class Tools

JAVA provides a nice tool-kit class called **Collections** that contains a bunch of useful methods that we can take advantage of. One of these is a **sort()** method which will sort an arbitrary collection.

Examine the following code to see how easy it is to sort our **ArrayList** of **Person** objects using this **sort()** method ...

```
import java.util.*;

public class SortTestProgram {
    public static void main(String args[]) {
        ArrayList<Person> people = new ArrayList<Person>();

        people.add(new Person("Pete Zaria", 12));
        people.add(new Person("Rita Book", 20));
        people.add(new Person("Willie Maykit", 65));
        people.add(new Person("Patty O'Furniture", 41));
        people.add(new Person("Sue Permann", 73));
        people.add(new Person("Sid Down", 19));
        people.add(new Person("Jack Pot", 4));
    }
}
```

```

        Collections.sort(people);           // do the sorting

        for (Person p: people)
            System.out.println(p);
    }
}

```

The output is as expected with all people sorted by their age.

Of course, we could use **AlphaPerson** to sort them alphabetical instead, if so desired. For the above code to work, we still need to have the **compareTo()** methods written. Hopefully you noticed how easy this **sort()** method is to use.

There is also a class called **Arrays** which has some useful methods for manipulating arrays. For example, if our code had arrays of **Person** objects instead of **ArrayLists**, here is what the code would look like to sort:

```

import java.util.*;

public class SortTestProgram2 {
    public static void main(String args[]) {
        Person[] people = {new Person("Pete Zaria", 12),
                           new Person("Rita Book", 20),
                           new Person("Willie Maykit", 65),
                           new Person("Patty O'Furniture", 41),
                           new Person("Sue Permann", 73),
                           new Person("Sid Down", 19),
                           new Person("Jack Pot", 4)};

        Arrays.sort(people);           // do the sorting

        for (Person p: people)
            System.out.println(p);
    }
}

```

There are similar sort methods for the primitive data types, so you can sort simple arrays of numbers such as this:

```

int[] nums = {23, 54, 76, 1, 29, 89, 45, 76};

Arrays.sort(nums);           // do the sorting

```

Interestingly, there are other useful methods in the **Collections** class such as **reverse()**, **shuffle()**, **max()** and **min()**. Can you guess what they do by looking at the output of the following program ?

```

import java.util.*;

public class SortTestProgram3 {
    public static void main(String args[]) {
        ArrayList<Person> people = new ArrayList<Person>();

        people.add(new Person("Pete Zaria", 12));
        people.add(new Person("Rita Book", 20));
        people.add(new Person("Willie Maykit", 65));
        people.add(new Person("Patty O'Furniture", 41));
        people.add(new Person("Sue Permann", 73));
        people.add(new Person("Sid Down", 19));
        people.add(new Person("Jack Pot", 4));

        System.out.println("The list reversed:");
        Collections.reverse(people);
        for(Person p: people)
            System.out.println(p);

        System.out.println("\nThe list shuffled:");
        Collections.shuffle(people);
        for(Person p: people)
            System.out.println(p);

        System.out.println("\nThe list shuffled again:");
        Collections.shuffle(people);
        for(Person p: people)
            System.out.println(p);

        System.out.println("\nOldest person: " + Collections.max(people));
        System.out.println("Youngest person:" + Collections.min(people));
    }
}

```



Here is the output ... was it as you expected? ...

```

The list reversed:
4 year old Jack Pot
19 year old Sid Down
73 year old Sue Permann
41 year old Patty O'Furniture
65 year old Willie Maykit
20 year old Rita Book
12 year old Pete Zaria

The list shuffled:
12 year old Pete Zaria
19 year old Sid Down
20 year old Rita Book
4 year old Jack Pot
65 year old Willie Maykit
41 year old Patty O'Furniture
73 year old Sue Permann

```

```

The list shuffled again:
65 year old Willie Maykit
20 year old Rita Book
4 year old Jack Pot
12 year old Pete Zaria
41 year old Patty O'Furniture
73 year old Sue Permann
19 year old Sid Down

```

```

Oldest person: 73 year old Sue Permann
Youngest person: 4 year old Jack Pot

```

There are additional methods in the **Collections** class. Have a look at the API and see if you find anything useful.

8.9 Implementing an ADT (Doubly-Linked Lists)

Consider allocating a large array of bytes:

```

byte[] myArray;

myArray = new byte[1000000];

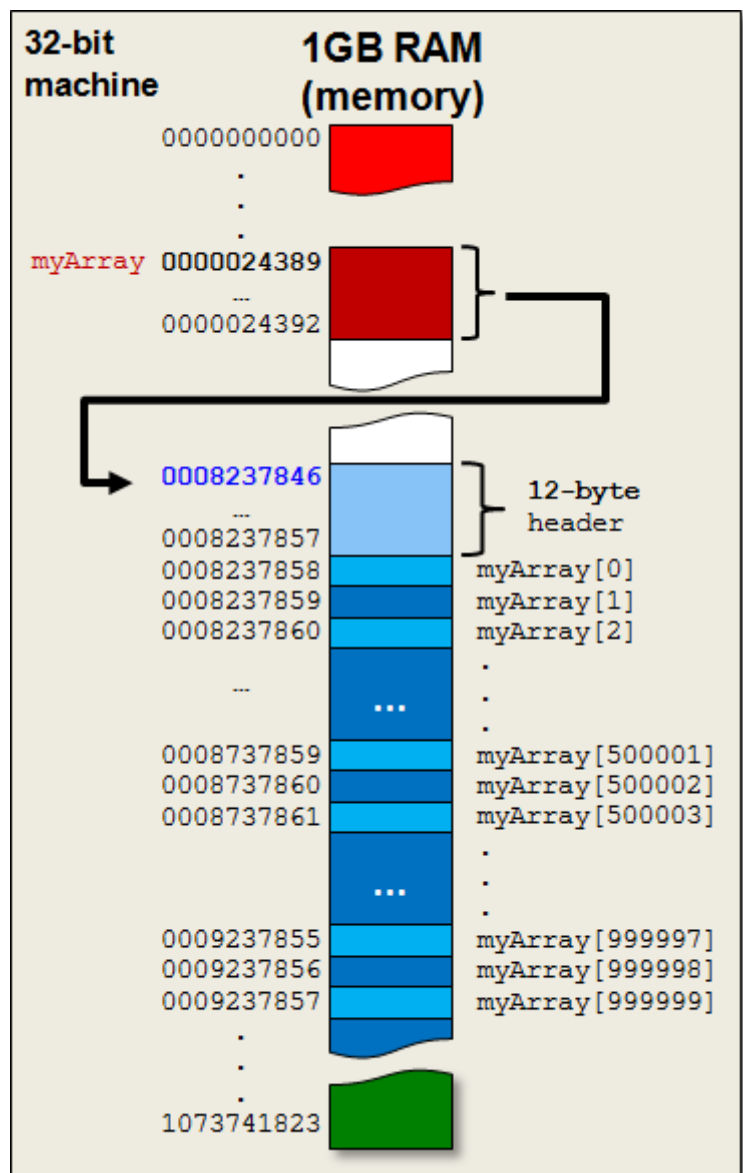
```

An array is an object and the values of the array are kept in consecutive memory locations. Usually the array length is also kept along with the array, so we have shown this as an extra **4 bytes** in the object header ... making it a **12-byte** header (although the exact size depends on the java implementation).

Assume that this array is filled with some appropriate values. Consider what happens when we want to remove the item at position 500002 in the array (shown red).

Remember that we cannot simply remove something from an array but that we have 2 choices:

1. replace the item at index 500002 with some clearly identifiable value (e.g., -1).
2. remove the item at that position by copying all the items from index 500003 to 999999 back 1 position in the array and then reduce the size of the array by 1.



Solution (1) is quick to do a remove operation, but then we will leave "gaps" in the array so that when we process it later we will need to consider the fact that there may be a lot of invalid data stored in the array at any time. In fact, after a while ... the array may be filled with mostly invalid data!

Solution (2) would take a lot more time to remove an item because we would potentially need to move large portions of the array back one position in memory each time we do a remove operation. In addition, we can "logically" reduce the array size by one, but in reality, JAVA has already allocated the memory for the 1,000,000 elements ... so that will not change. In other words, we are essentially classifying the "end portion" of the array as **garbage data** as time goes on. We are not saving any space ... the garbage/wasted data is still taking up memory.

This problem gets worse as we consider adding items to the array beyond the 1,000,000 capacity. In that case, we would need to create a whole new bigger array and copy all the items from the "old" array into the "new" array ... then discard the old array. To do this, we would simply move the **myArray** variable pointer to point to the new array:

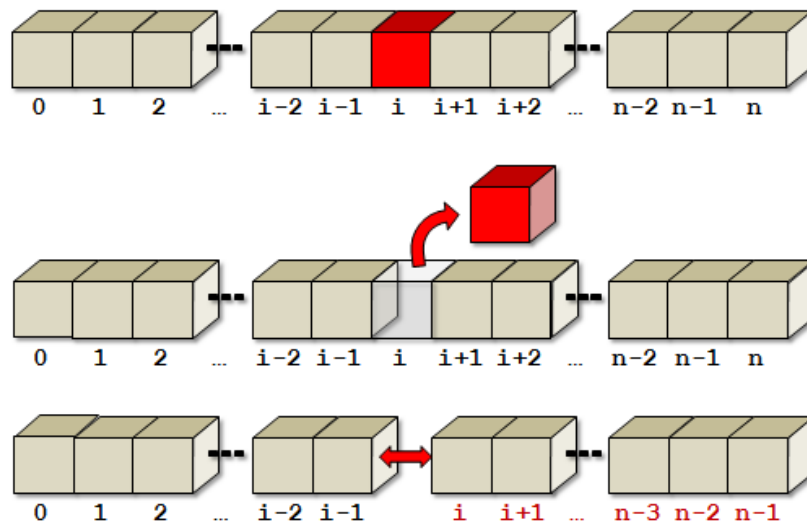
```
myArray = new byte[2000000];
```

This would create a whole new object which takes another 2,000,000 bytes (+12 for header). The Java VM would then realize that the data in memory from address 0008237846 to 0009237857 would no longer be needed and it would be scheduled for a future garbage collection operation. In languages such as C or C++, there is no garbage collector, so we would have to remember to free up that memory on our own.

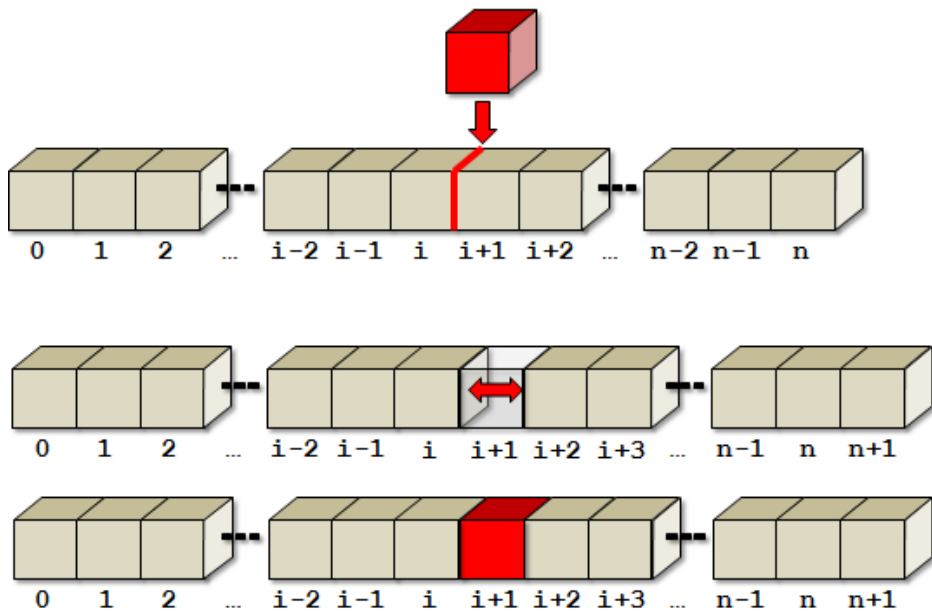
One huge danger of this "new-array-reallocation-and-copy-over" strategy is that if any other objects are pointing to the old array ... then it is not garbage collected and potentially we have two places in our code that at one time may have been pointing to the same array but are now pointing to different arrays !!

One solution to this problem is to store data in what is called a **doubly-linked list**. We would like to be able to do two things:

1. Cut out a single piece of data and stitch the remaining data back together:



2. Cut open a spot in the data and insert a single piece of data inside:



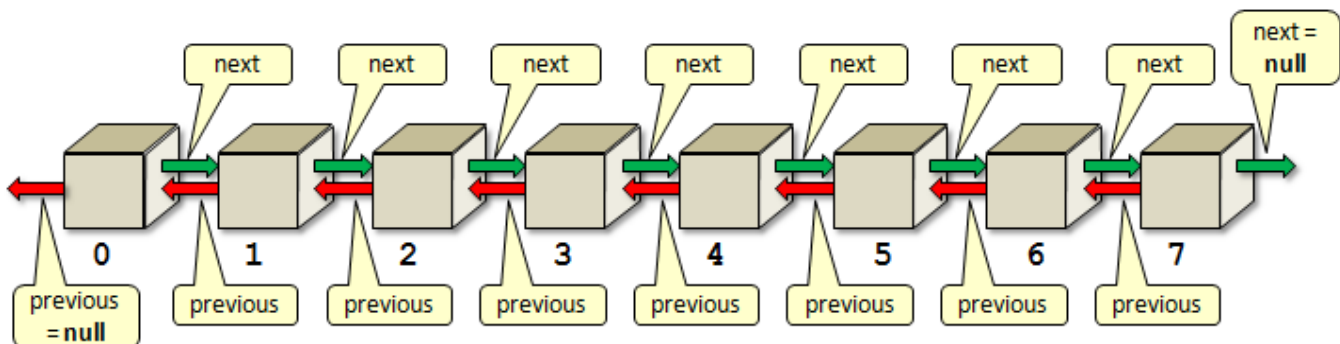
To do this, we need to allow the data to be split and merged anywhere within the list of data. We can do this by allowing each piece of data to be its own object. As long as each of these objects knows the object before it in the list as well as the object after it in the list, then we can make this happen. Consider a single item in the list represented as follows:

```

public class Item {
    byte data;
    Item previous;
    Item next;

    public Item(int d) {
        data = (byte)d;
        previous = null;
        next = null;
    }
}
    
```

Notice that this **Item** class represents a *recursive data structure* definition since the item before (i.e., *previous*) this item is an **Item** object and the item after (i.e., *next*) it is also an **Item** object. That means, the items each keep a pointer to the object before it in the list and the object after it in the list. So we can re-draw our **n-item** list now as follows:



Consider a simple array created as follows:

```
byte[] myList = new byte[8];
myList[0] = 23; myList[1] = 65;
myList[2] = 87; myList[3] = 45;
myList[4] = 56; myList[5] = 34;
myList[6] = 95; myList[7] = 71;
```

Here is how we would create the linked-list version for this list of data:

```
Item myList = new Item(23);
Item myList1 = new Item(65);
Item myList2 = new Item(87);
Item myList3 = new Item(45);
Item myList4 = new Item(56);
Item myList5 = new Item(34);
Item myList6 = new Item(95);
Item myList7 = new Item(71);

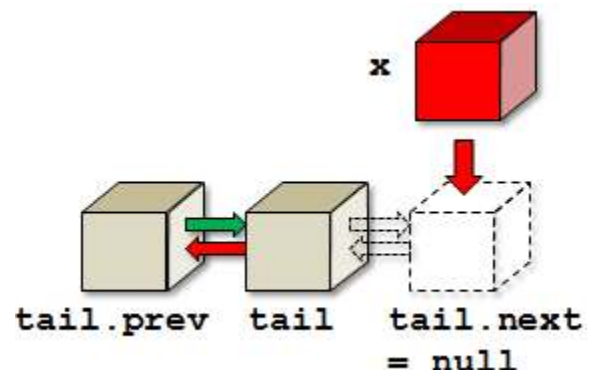
myList.previous = null;
myList.next = myList1;
myList1.previous = myList;
myList1.next = myList2;
myList2.previous = myList1;
myList2.next = myList3;
myList3.previous = myList2;
myList3.next = myList4;
myList4.previous = myList3;
myList4.next = myList5;
myList5.previous = myList4;
myList5.next = myList6;
myList6.previous = myList5;
myList6.next = myList7;
myList7.previous = myList6;
myList7.next = null;
```

This code is a bit ugly because it uses many variable names. However, typically we would create operations for adding and removing Items. Also, we usually want to keep track of the **first** and **last** items in the linked list ... which are known as the **head** and the **tail**. So, often we create another class to keep track of this information as follows:

```
public class LinkedList {
    Item head;
    Item tail;

    public LinkedList() {
        head = null;
        tail = null;
    }
}
```

Then we would make operations in this class. One useful operation would be to add an item to the end (i.e., tail) of the list.



Here is the code that will do this:

```
public void add(Item x) {
    if (tail == null) {
        tail = x;
        head = x;
    }
    else {
        tail.next = x;
        x.previous = tail;
        tail = x;
    }
}
```

Notice that we had to handle the case where we called the method the very first time. In that case, the head and the tail would both be **null**. So, when adding in that case, the item being added becomes the sole item in the list ... making it both the head and the tail at the same time. From then on, all additions occur at the tail end of the list.

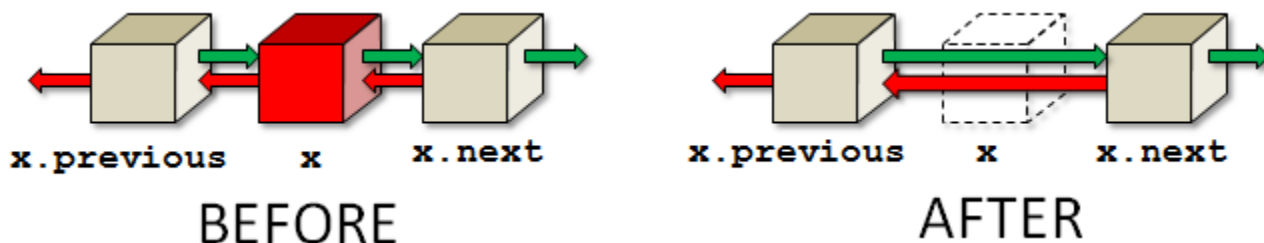
Once we have this method available, the code to construct the list becomes simplified:

```
LinkedList myList = new LinkedList();
myList.add(new Item(23));
myList.add(new Item(65));
myList.add(new Item(87));
myList.add(new Item(45));
myList.add(new Item(56));
myList.add(new Item(34));
myList.add(new Item(95));
myList.add(new Item(71));
```

Is this better than an array? It seems like a lot of overhead! Well ... it may indeed take up more space ... but the size of the list is unlimited (except for running out of computer memory). That is ... we never have to worry about going past an array bounds. Also, we never have to worry about re-allocating a new larger array and copying elements over into it.

What about the removal of an item from the list?

It too just involves moving a couple of pointers around.



Here is the code to remove an item ... assuming that **x** is in the list:

```

public void remove(Item x) {
    if (x == head) {
        if (x == tail) {
            head = tail = null;
        }
        else {
            head = x.next;
            head.previous = null;
        }
    }
    else {
        if (x == tail) {
            tail = x.previous;
            tail.next = null;
        }
        else {
            x.previous.next = x.next;
            x.next.previous = x.previous;
        }
    }
}

```

The code looks a little long because we need to handle the special cases in which the removed item is the **head** of the list or the **tail** of the list. However, you will notice that the code for removal simply involves the changing of two pointers. There is no need to copy items back in the array, nor is there any concern about garbage data lying around. The code is quite simple and elegant.

How could we write a **toString()** method for this list that shows the contents? Assume that we want the list to look like this:

[H:23]<==>[65]<==>[87]<==>[45]<==>[56]<==>[34]<==>[95]<==>[71:T]

or like this when 1 item is in it: [H:23:T]
 or like this when empty: [EMPTY]

To iterate through the items, we would need to start with the **head** of the list and keep traversing successive **.next** pointers until we reached the **tail**.

```

public String toString() {
    if (head == null)
        return "[EMPTY]";

    String s = "[H:";
    Item currentItem = head;
    while (currentItem != null) {
        s += currentItem.data;
        if (currentItem != tail)
            s += "<==>[";
        currentItem = currentItem.next;
    }
    return s + ":T]";
}

```

How would we write a method to add up all of the byte data in the list? It is quite similar:

```
public int totalData() {
    if (head == null)
        return 0;

    int total = 0;
    Item currentItem = head;
    while (currentItem != null) {
        total += currentItem.data;
        currentItem = currentItem.next;
    }
    return total;
}
```

Do you understand why a **for** loop was not used ?

Here is a test program:

```
public class LinkedListTestProgram {
    public static void main(String args[]) {
        Item head, tail, internal;

        LinkedList myList = new LinkedList();
        myList.add(head = new Item(23));
        myList.add(new Item(65));
        myList.add(new Item(87));
        myList.add(internal = new Item(45));
        myList.add(new Item(56));
        myList.add(new Item(34));
        myList.add(new Item(95));
        myList.add(tail = new Item(71));

        System.out.println("Here is the list: ");
        System.out.println(myList);

        System.out.println("\nThe total of the data is: ");
        System.out.println(myList.totalData());

        System.out.println("\nRemoving the head .. here is the list now: ");
        myList.remove(head);
        System.out.println(myList);

        System.out.println("\nRemoving the tail .. here is the list now: ");
        myList.remove(tail);
        System.out.println(myList);

        System.out.println("\nRemoving internal item 45, here is the list now: ");
        myList.remove(internal);
        System.out.println(myList);
    }
}
```

Here is the output:

```

Here is the list:
[H:23]<==>[65]<==>[87]<==>[45]<==>[56]<==>[34]<==>[95]<==>[71:T]

The total of the data is:
476

Removing the head .. here is the list now:
[H:65]<==>[87]<==>[45]<==>[56]<==>[34]<==>[95]<==>[71:T]

Removing the tail .. here is the list now:
[H:65]<==>[87]<==>[45]<==>[56]<==>[34]<==>[95:T]

Removing internal item 45, here is the list now:
[H:65]<==>[87]<==>[56]<==>[34]<==>[95:T]

```

Can you write an **insert(x, i)** method that will insert item x after position i in the list ? Try it. You will need to start at the head of the list and count past i items before you start changing pointers. Can you do a **remove(i)** method that will remove the i'th item from the list ?

It is important to understand how to manipulate pointers like this because some languages (e.g., C and C++) require a lot of memory allocation and pointer manipulation. The more practice you get ... the better!!

You should realize that although our list contained simple data in the form of a single **byte**, you can simply change the type of the **data** to any data type. In this way, the list can store any kind of data that you want. Here is a general definition for a list Item that can store any object:

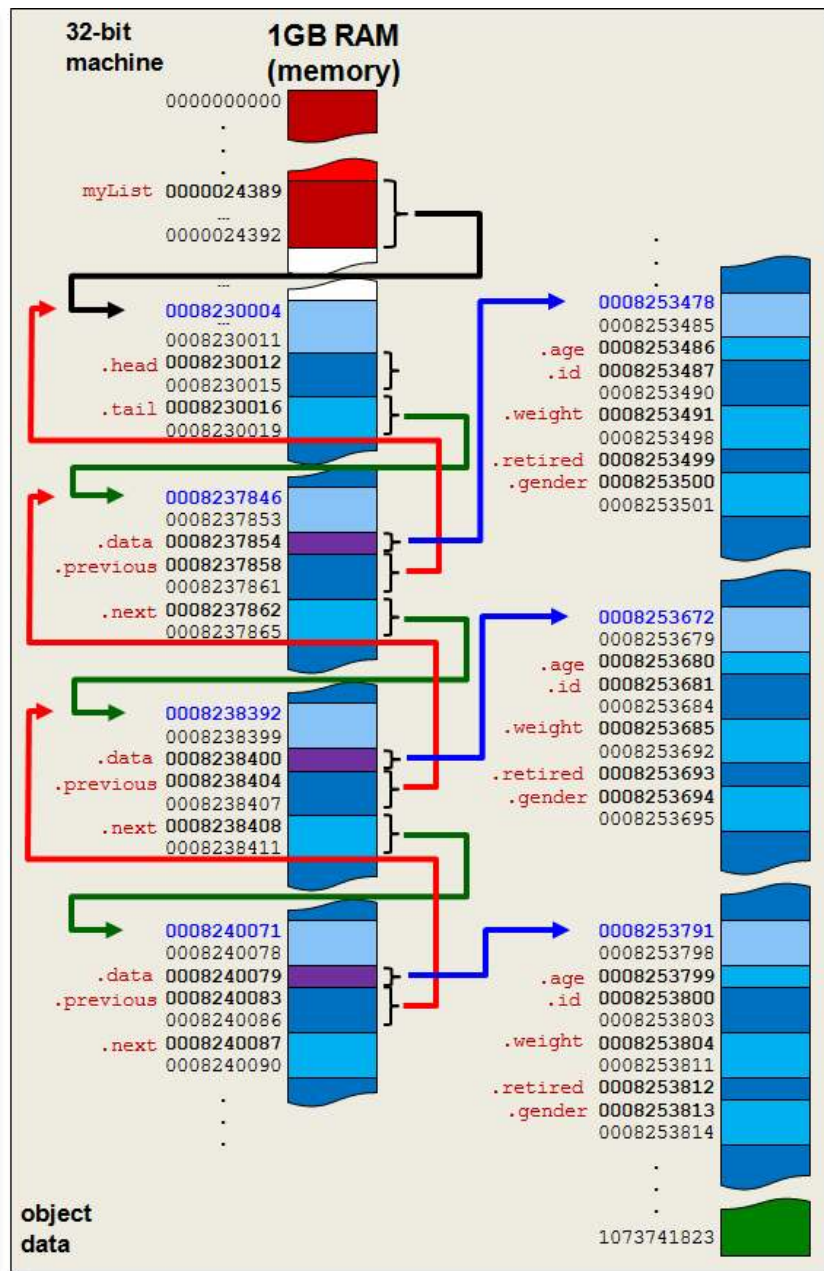
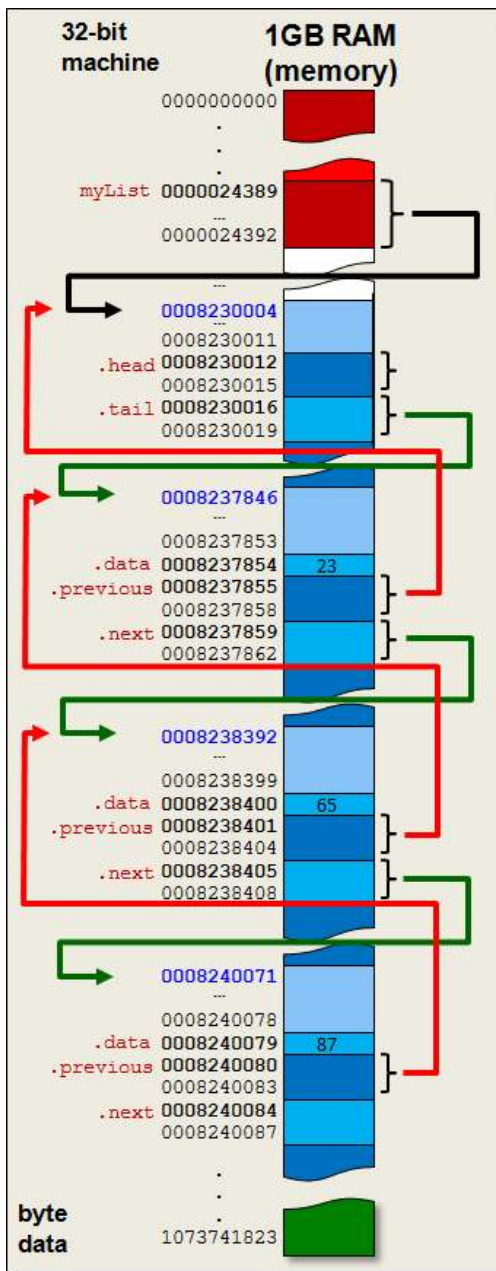
```

public class Item {
    Object data;
    Item previous;
    Item next;

    public Item(Object obj) {
        data = obj;
        previous = null;
        next = null;
    }
}

```

Notice what the memory allocation would look like for a simple 3 item list when simple **byte** data is used (left side diagram) and when **Person** object data is used (right side diagram):



Chapter 9

Recursion With Data Structures

What is in This Chapter ?

In the last course, we discussed recursion at a simple level. This chapter explains how to do more complex *recursion* using various data structures. You should understand recursion more thoroughly after this chapter.



9.1 Recursive Efficiency

You should already be familiar with recursion at this point, having taken it last term in COMP1405. Although recursion is a powerful problem solving tool, it has some drawbacks. A non-recursive (or iterative) method may be **more** efficient than a recursive one for two reasons:

1. there is an overhead associated with large number of method calls
2. some algorithms are inherently inefficient.

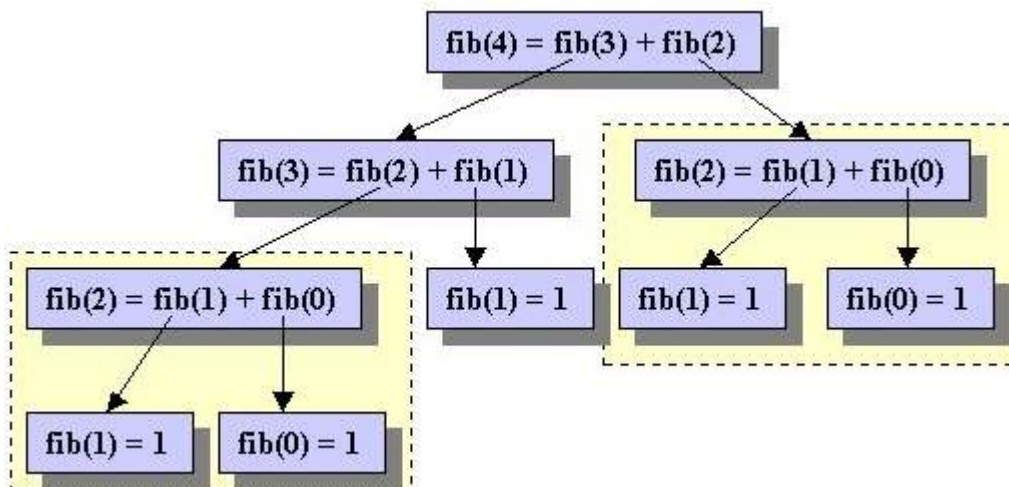
For example, computing the n th Fibonacci number can be written as:

1	if $n = 0$
fib(n) = 1	if $n = 1$
fib(n-1) + fib(n-2)	if $n > 1$

A straight-forward recursive solution to solving this problem would be as follows:

```
public static int fibonacci(int n) {
    if (n <= 1)
        return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

However, notice what is happening here:



In the above computation, some problems (e.g., **fibonacci(2)**) are being solved more than once, even though we presumably know the answer after doing it the first time. This is an example where recursion can be inefficient if we do not do it carefully.

The following iterative solution avoids the re-computation:


```

public static int fibonacci2(int n) {
    int first = 1;
    int second = 1;
    int third = 1;
    for (int i=2; i<=n; i++) {
        third = first + second; // compute new value
        first = second;         // shift the others to the right
        second = third;
    }
    return third;
}

```

Notice how the previous two results are always stored in the **first** and **second** variables. So the computation need not be duplicated. We can do this recursively ... we just need to keep track of previous computations.

```

public static int fibonacci(int n, int prev, int prevPrev) {
    if (n <= 0)
        return prev + prevPrev;
    else
        return fibonacci(n-1, prev + prevPrev, prev);
}

```

We essentially use the same idea of keeping track of the last two computations and passing them along to the next recursive call. However, we would have to start this off with some values for these parameters. For example to find the 20th Fibonacci number, we could do this:

```

    fibonacci(18, 1, 1);

```

The first two numbers are 1 and then there are 18 more to find.

However, this is not a nice solution because the user of the method must know what the proper values are in order to call this method. It would be wise to make this method **private** and then provide a **public** one that passes in the correct initial parameters as follows:

```

public static int fibonacci3(int n) {
    if (n <= 1)
        return 1;
    return fibonacci(n-2, 1, 1); // calls above method
}

```

This method in itself is not recursive, as it does not call itself. However, indirectly, it does call the 3-parameter **fibonacci()** method ... which is recursive. We call this kind of method indirectly recursive.

*An **indirectly recursive** function is one that does not call itself, but it does call a recursive method.*

Indirect recursion is mainly used to supply the initial parameters to a recursive function. It is the one that the user interacts with. It is often beneficial to use recursion to improve efficiency as well as to create non-destructive functions.

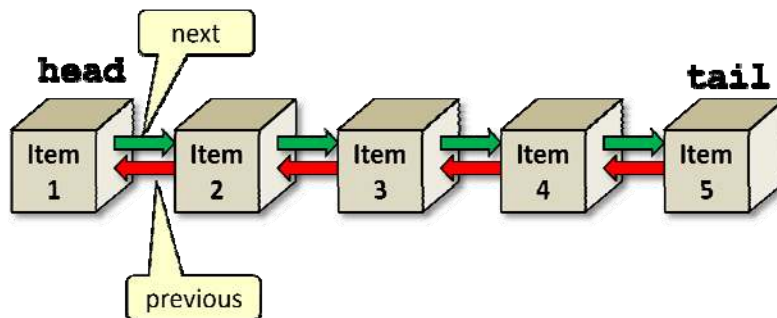
9.2 Examples With Self-Referencing Data Structures

Until now, the kinds of problems that you solved recursively likely did not involve the use of data structures. We will now look at using recursion to solve problems that make use of a couple of simple data structures.

First, recall the linked-list data structure that we created in the last chapter. It is a self-referencing data structure since each **Item** object points to two other **Item** objects:

```
public class LinkedList {
    Item head;
    Item tail;
    ...
}
```

```
public class Item {
    byte data;
    Item previous;
    Item next;
    ...
}
```



We can write some interesting recursive methods for this data structure.

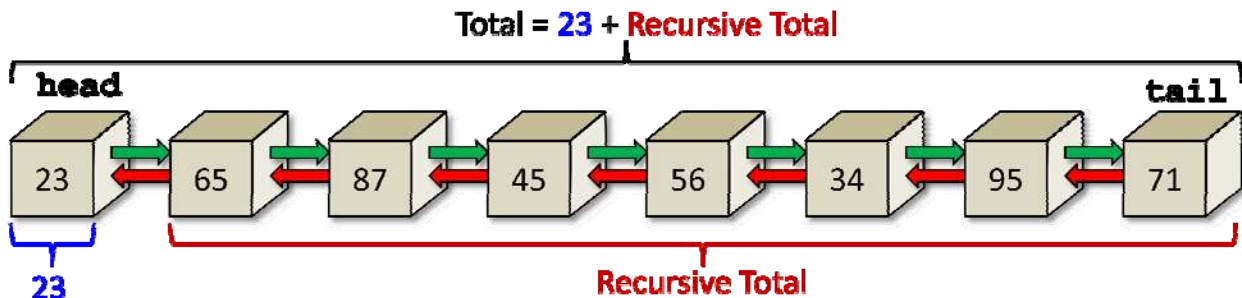
Example:

Recall the following method:

```
public int totalData() {
    if (head == null)
        return 0;

    int total = 0;
    Item currentItem = head;
    while (currentItem != null) {
        total += currentItem.data;
        currentItem = currentItem.next;
    }
    return total;
}
```

Let us see how we can write this method recursively without using a **while** loop. As before, we need to consider the base case. If the **head** of the list is **null**, then the answer is **0** as in the above code. Otherwise, we will need to break the problem down recursively. To do this, we can simply break off the first data item from the list and add it to the recursive result of the remainder of the list ... this will be the solution:



However, we do not want to destroy the list, so we will simply "pretend" to break off a piece by traversing through the **next** pointers of the items, starting at the **head**. The solution is straightforward as long as we are allowed to pass in a parameter representing the item in the list from which to start counting from (e.g., the head to begin). Recall, that we always begin with a "base case" ... which is the stopping condition for the recursion. It always represents the simplest situation for the data structure. In this case, the simplest case is a **null** item. If the item is null, there are no numbers to add, so the result is clearly 0. Otherwise, we just need to "tear off" the first number and continue with the remainder of the list (i.e., continue adding ... but starting with the **next** item in the list):

```
private int totalDataRecursive(Item startItem) {
    if (startItem == null)
        return 0;

    return startItem.data + totalDataRecursive(startItem.next);
}
```

Notice the simplicity of the recursion. It is quite straight forward and logical. As it turns out, writing recursive methods for most self-referencing data structures is quite natural and often produces simple/elegant code.

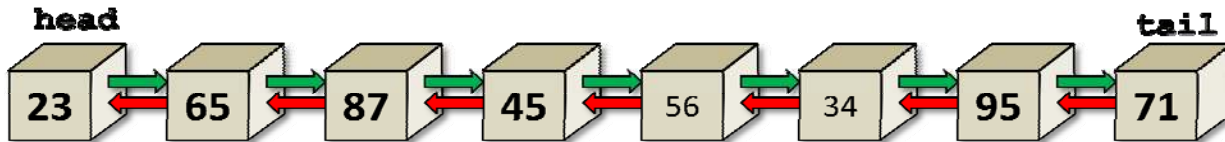
One downfall of the above method is that it requires a parameter which **MUST** be the **head** item of the list if it is to work properly! So then, we will want to make a **public** method that the user can call which will create the proper starting parameter (i.e., the **head**):

```
public int totalDataRecursive() {
    return totalDataRecursive(head);
}
```

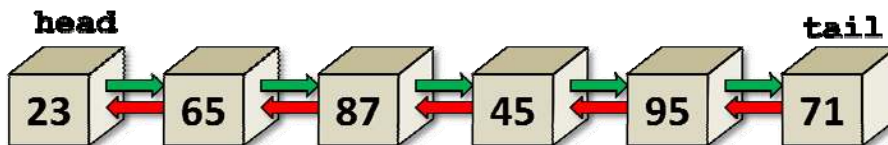
As you can see, the code is quite short. It simply supplies the list **head** to the single-parameter recursive method. This method is not itself recursive, but it does call the single-parameter method which IS recursive. We therefore call this an **indirectly recursive** method because although the method itself is not recursive, the solution that the method provides is recursive.

Example:

Now what about writing a recursive method that returns a new **LinkedList** that contains all the *odd* data from the list ?



The method should return a new **LinkedList**:



Hence, the return type for the method should be **LinkedList**.

```
public LinkedList oddItems() {
    // ...
}
```

The method code should begin with a "base case". What is the simplest list that we can have for use in the "base case" ? An empty one (i.e., headless), of course!

```
public LinkedList oddItems() {
    if (head == null)
        return new LinkedList();
    // ...
}
```

Otherwise, we will need to apply the same strategy of breaking off a piece of the problem. We can do this using indirect recursion again by starting with the head. It will be easiest, as well to have the new list passed in as a parameter that we can simply add to:

```
public LinkedList oddItems() {
    if (head == null)
        return new LinkedList();

    return oddItems(head, new LinkedList());
}
```

So then, the directly-recursive method will be defined as follows:

```
private LinkedList oddItems(Item startItem, LinkedList resultList) {
    // ...
}
```

Notice that it is private, because it is just a kind of *helper* method for the public one.

Since the **startItem** will eventually become **null**, we will need to check for that as our stopping condition (i.e., "base case"). In that case, we are done ... and we just need to return the **resultList**:

```
private LinkedList oddItems(Item startItem, LinkedList resultList) {
    if (startItem == null)
        return resultList;
    // ...
}
```

As a side point, since we are checking for **null** here in this method as well, we can go back and simplify our indirectly-recursive method by removing that check:

```
public LinkedList oddItems() {
    return oddItems(head, new LinkedList());
}
```

Now, we need to check to see if the data of the **startItem** is indeed odd, and if so... then add it to the **resultList**:

```
private LinkedList oddItems(Item startItem, LinkedList resultList) {
    if (startItem == null)
        return resultList;

    if (startItem.data %2 != 0)
        resultList.add(new Item(startItem.data));
    // ...
}
```

Notice that we used a constructor to create a new **Item** object before adding to the resulting list. What would have happened if we simply used `resultList.add(startItem)`? If we did this, then the same **Item** object would exist in both lists. This is bad because if we changed the **next** or **previous** pointer for this item, then it would affect both lists... and that would be disastrous.

Lastly, we continue by checking the remainder of the list:

```
private LinkedList oddItems(Item startItem, LinkedList resultList) {
    if (startItem == null)
        return resultList;

    if (startItem.data %2 != 0)
        resultList.add(new Item(startItem.data));

    return oddItems(startItem.next, resultList);
}
```

Do you understand why the **return** keyword is needed on the last line? What gets returned? Well ... remember that the method **MUST** return a **LinkedList**. If we leave off the **return** keyword, the compiler will complain because we are not returning any specific list. At the end of the recursive method calls, the "base case" will ensure that we return the **resultList** that we have built up with the odd numbers. Alternatively we could have just finished off the recursion and then specifically state that we want to return the **resultList**:

```

private LinkedList oddItems(Item startItem, LinkedList resultList) {
    if (startItem == null)
        return resultList;

    if (startItem.data %2 != 0)
        resultList.add(new Item(startItem.data));

    oddItems(startItem.next, resultList);

    return resultList;
}

```

We can actually make this all work without that extra **resultList** parameter by creating the list when we reach the end of the list, and then adding items AFTER the recursion:

```

public LinkedList oddItems() {
    return oddItems(head);
}

private LinkedList oddItems(Item startItem) {
    if (startItem == null)
        return new LinkedList();

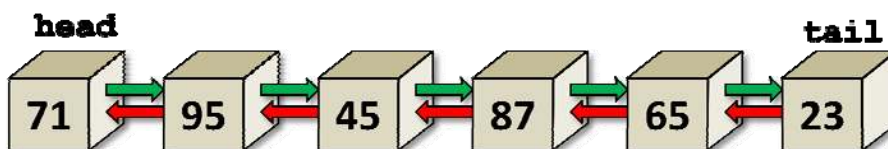
    LinkedList result = oddItems(startItem.next);

    if (startItem.data %2 != 0)
        result.add(new Item(startItem.data));

    return result;
}

```

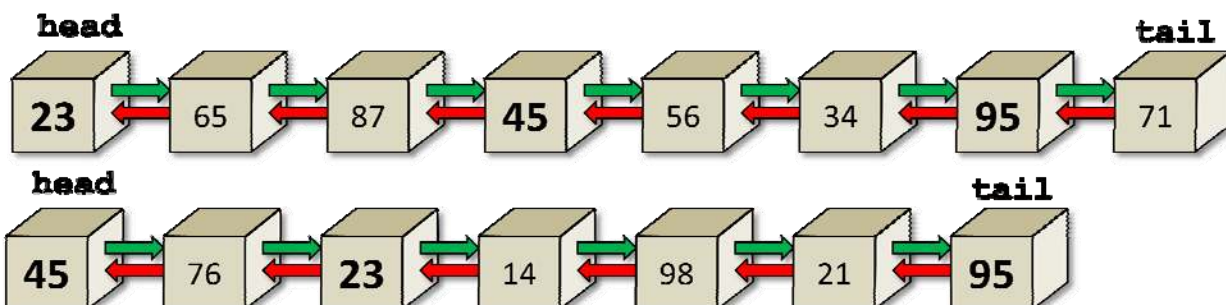
However, this solution will return the odd numbers in reverse order.



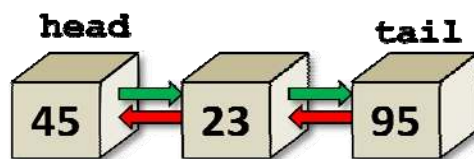
See if you can fix that problem. It can be done.

Example:

Now, let us find and return a list of all elements in common between 2 lists:



Again, the result will be a new **LinkedList**:



To do this, we will need a similar indirectly-recursive method, but one that takes the 2nd list as a parameter:

```

public LinkedList inCommon(LinkedList aList) {
    return inCommon(this.head, aList.head, new LinkedList());
}
  
```

Here, we see that the two list **heads** are passed in as well as a list onto which the common elements may be added. The directly-recursive method begins the same way as before ... quitting when we reach the end of one of the lists:

```

private LinkedList inCommon(Item start1, Item start2, LinkedList result) {
    if ((start1 == null) || (start2 == null))
        return result;
    // ...
}
  
```

The remaining problem is a bit trickier. We need to check each item in one list with each item in the other list ... but just once. So we will need to recursively shrink one list until it has been checked against one particular item in the second list. Then, we need to move to the next item in the second list and check it against the entire first list again. It will be easier to do this if we had a *helper* method that simply checked whether or not a data item is in another list.

Consider a method called **contains()** which takes a starting item in the list and determines whether or not a specific data item is in the list by recursively iterating through the list. Can you write this? It should be straight forward now:

```

public boolean contains(Item startItem, byte data) {
    if (startItem == null)
        return false;

    if (startItem.data == data)
        return true;

    return contains(startItem.next, data);
}
  
```

Now, how can we make use of this **contains()** method to solve our original problem? Well, we can call it like any other function. We can simply iterate through the items of one list (as before) and check each item against the other list using this **contains()** function. If it IS contained, we add it to the solution. It is quite similar to the template for finding the odd numbers now:

```

private LinkedList inCommon(Item list1, Item list2, LinkedList result) {
    if ((list1 == null) || (list2 == null))
        return result;

    // Check if the first list contains the data
    // at the beginning item of the 2nd list.
    if (contains(list1, list2.data))
        result.add(new Item(list2.data));

    // Now check the first list with the remainder of the 2nd list
    return inCommon(list1, list2.next, result);
}

```

As you can see, the call to **contains()** here will check all items of list 1 with the first item of list 2. Then, the recursive call will move on to the next item in list 2. Eventually, list 2 will be exhausted and we will have the solution!

Example:

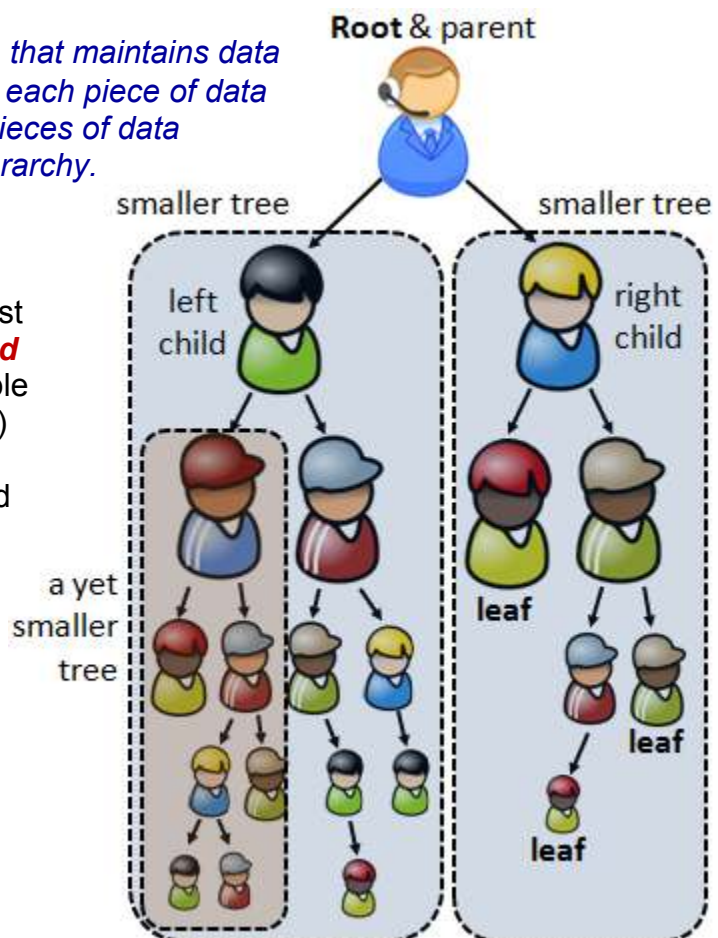
Now let us try a different data structure. In computer science, we often store information in a **binary tree**:

A **binary tree** is a data structure that maintains data in a hierarchical arrangement where each piece of data (called the **parent**) has exactly two pieces of data (called **children**) beneath it in the hierarchy.

The binary tree is similar to the notion of a single gender (i.e., all males or all females) family tree in which every parent has at most 2 children, which are known as the **leftChild** and **rightChild** of their **parent**. It is possible that a **node** in the tree (i.e., a piece of data) may have no children, or perhaps only one child. In this case, we say that the leftChild and/or rightChild is **null** (meaning that it is non-existent).

A node with no children is called a **leaf** of the tree.

The **root** of the tree is the data that is at the top of the tree which has no parent.

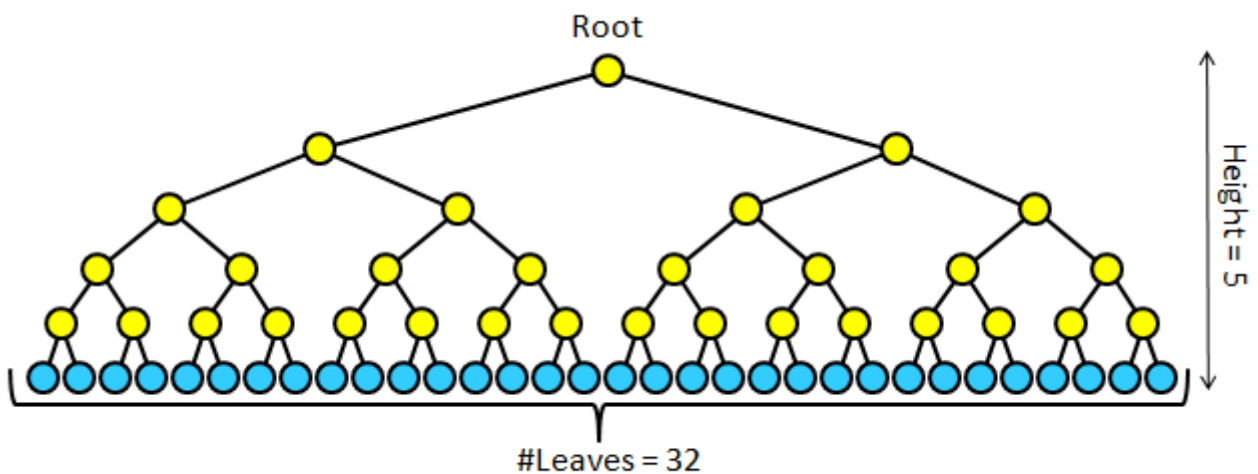


Binary trees can be a very efficient data structure when searching for information. For example as we search for a particular item down the tree from the root, each time that we choose the left or right child to branch down to, we are potentially eliminating half of the remaining data that we need to search through.

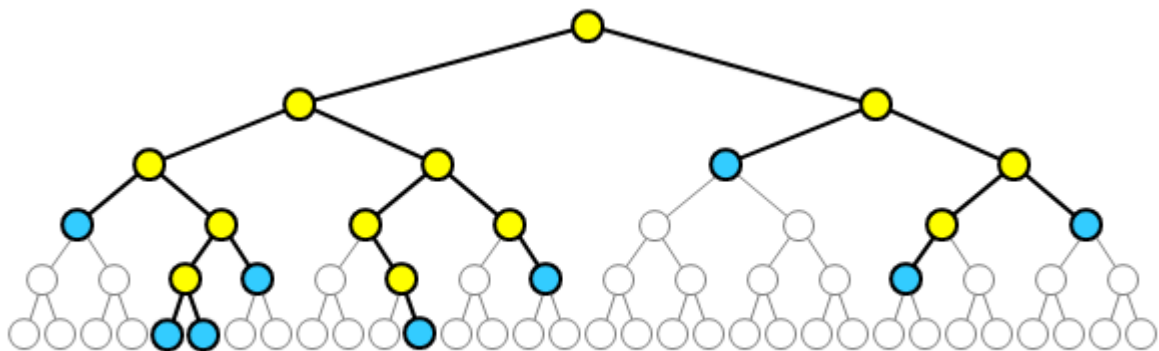
Typically, binary trees are represented as recursive data structures. That is, the tree itself is actually made up of other smaller trees. We can see this from the figure on the previous page, where each non-**null** child actually represents the root of a smaller tree.

In computer science, trees are often drawn with simple circles as nodes and lines as edges.

The **height** of a tree is the depth of the tree from root to the leaves. Here is an example of a **complete** tree (i.e., one that is completely filled with nodes) of height 5. A complete binary tree has 2^h leaves (where h is the tree's height).



In our family tree picture (shown earlier), however, the tree was not complete ... there were nodes missing. Here is how we would draw the equivalent tree for that example:



Notice that it is basically the 2^5 binary tree with many of the nodes removed. Notice as well, that the leaves are not only at the bottom level of the tree but may appear at any level because any node that has no children is considered a leaf.

How can we represent a recursive data structure like this binary tree? Well, remember, each time we break a branch off of a tree, we are left with two smaller trees. So a binary tree itself is made up of two smaller binary trees. Also, since trees are used to store data, each node of the tree should store some kind of information. Therefore, we can create a **BinaryTree** data structure as follows:

```
public class BinaryTree {
    private String      data;
    private BinaryTree leftChild;
    private BinaryTree rightChild;
}
```

Here, **data** represents the information being stored at that node in the tree ... it could be a String, a number, a Point, or any data structure (i.e., Object) with a bunch of information stored in it. Notice that the left **leftChild** and **rightChild** are actually binary trees themselves! A tree is therefore considered to be a *self-referential* (i.e., refers to itself) data structure and is thus a naturally recursive data structure.

Likely, we will also create some constructors as well as some get/set methods in the class:

```
public class BinaryTree {
    private String      data;
    private BinaryTree leftChild;
    private BinaryTree rightChild;

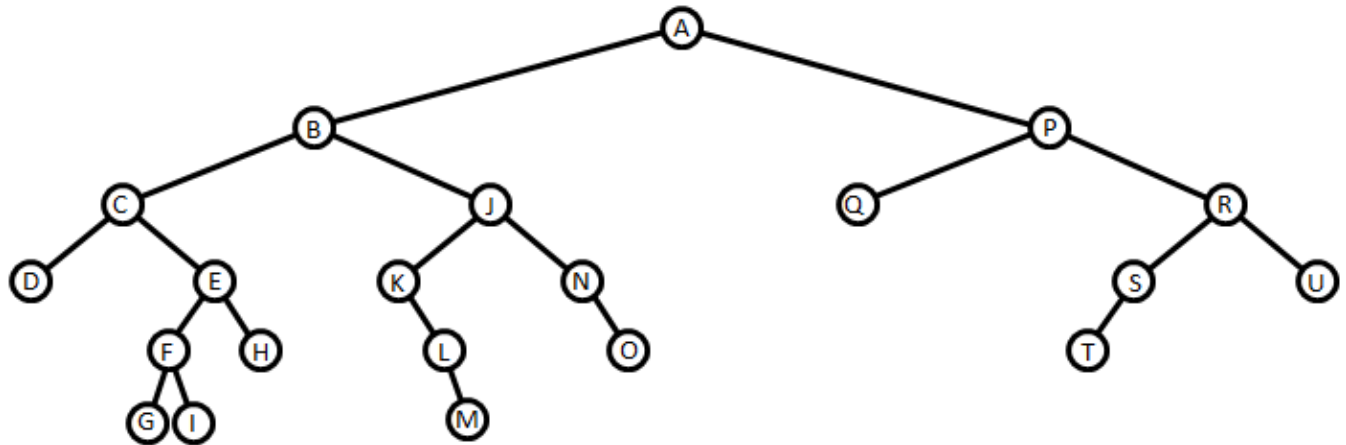
    // A constructor that takes root data only and
    // makes a tree with no children (i.e., a leaf)
    public BinaryTree(String d) {
        data = d;
        leftChild = null;
        rightChild = null;
    }

    // A constructor that takes root data as well as two subtrees
    // which then become children to this new larger tree.
    public BinaryTree(String d, BinaryTree left, BinaryTree right) {
        data = d;
        leftChild = left;
        rightChild = right;
    }

    // Get methods
    public String getData() { return data; }
    public BinaryTree getLeftChild() { return leftChild; }
    public BinaryTree getRightChild() { return rightChild; }

    // Set methods
    public void setData(String d) { data = d; }
    public void setLeftChild(BinaryTree left) { leftChild = left; }
    public void setRightChild(BinaryTree right) { rightChild = right; }
}
```

To create an instance of **BinaryTree**, we simply call the constructors. Consider a tree with the data for each node being a simple string with a letter character as follows:



Here is a test program that creates this tree:

```

public class BinaryTreeTest {
    public static void main(String[] args) {
        BinaryTree root;

        root = new BinaryTree("A",
            new BinaryTree("B",
                new BinaryTree("C",
                    new BinaryTree("D"),
                    new BinaryTree("E",
                        new BinaryTree("F",
                            new BinaryTree("G"),
                            new BinaryTree("I")),
                        new BinaryTree("H"))),
                new BinaryTree("J",
                    new BinaryTree("K",
                        null,
                        new BinaryTree("L",
                            null,
                            new BinaryTree("M"))),
                    new BinaryTree("N",
                        null,
                        new BinaryTree("O")))),
            new BinaryTree("P",
                new BinaryTree("Q"),
                new BinaryTree("R",
                    new BinaryTree("S",
                        new BinaryTree("T"),
                        null),
                    new BinaryTree("U"))));
    }
}

```

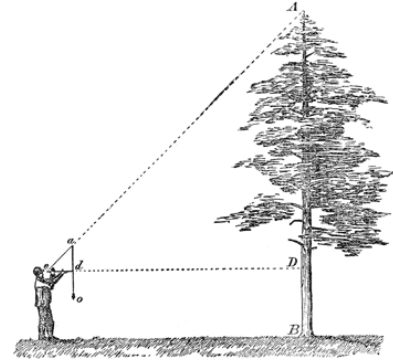
Example:

When we have such recursive data structures, it is VERY natural to develop recursive functions and procedures that work with them. For example, consider finding the height of this tree. It is not natural to use a **FOR** loop because we have no array or list to loop through.

How can we write a **recursive** function that determines the height of a binary tree ?

We need to determine the base case(s). What is the simplest tree ? It is one where the children are **null** (i.e., just a root). In this case, the height is 0.

Here is the code so far as an instance method in the **BinaryTree** class:



```
public int height() {
    if ((leftChild == null) && (rightChild == null))
        return 0;
}
```

That was easy. Now, for the recursive step, we need to express the height of the tree in terms of the smaller trees (i.e., its children). So, if we knew the height of the **leftChild** and the height of the **rightChild**, how can we determine the height of the "whole" tree ?

Well, the height of the tree is one more than the trees beneath it. Assuming that the left and right sub-trees are equal in height, the recursive definition would be:

$$h(\text{tree}) = 1 + h(\text{tree.leftChild})$$

However, as you can see from our family tree example, it is possible that the left and right children will have different heights (i.e., 4 and 3 respectively). So, to find the height of the whole tree, we need to take the largest of these sub-trees. So here is our recursive definition:

$$h(\text{tree}) = 1 + \text{maximum}(h(\text{tree.leftChild}), h(\text{tree.rightChild}))$$

Here is the code:

```
public int height() {
    if ((leftChild == null) && (rightChild == null))
        return 0;
    return 1 + Math.max(leftChild.height(),
                        rightChild.height());
}
```

However, there is a slight problem. If one of the children is **null**, but not the other, then the code will likely try to find the **leftChild** or **rightChild** of a **null** tree ... and this will generate a **NullPointerException** in our program. We can fix this in one of two ways:

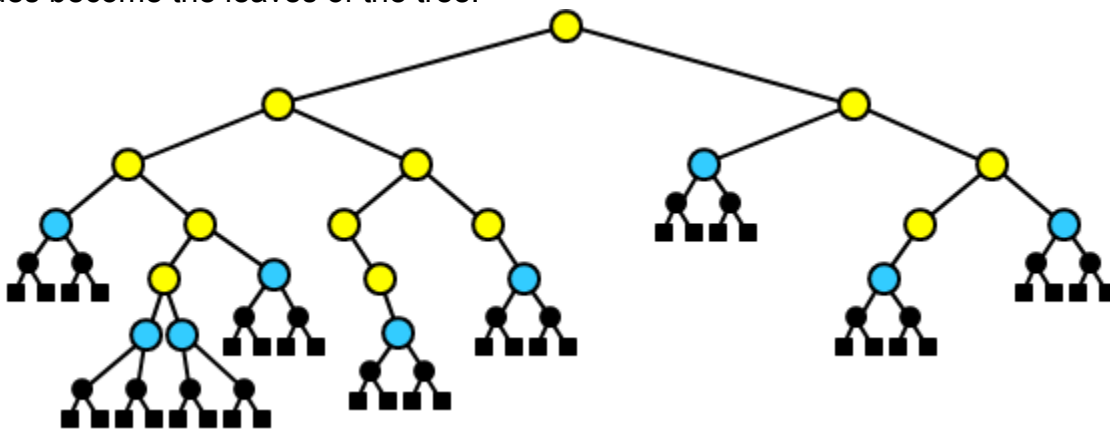
- (1) check for the case where one child is **null** but not the other
- (2) handle **null** trees as a base case.

Here is the solution for (1):

```
public int height() {
    if (leftChild == null) {
        if (rightChild == null)
            return 0;
        else
            return 1 + rightChild.height();
    }
    else {
        if (rightChild == null)
            return 1 + leftChild.height();
        else
            return 1 + Math.max(leftChild.height(),
                                rightChild.height());
    }
}
```

The above code either checks down one side of the tree or the other when it encounters a tree with only one child. If there are no children, it returns **0**, and otherwise it takes the maximum of the two sub-trees as before.

In choice (2) for dealing with **null** children, it is simpler just to add a base-case for handling **null** tree roots. However this requires the addition of extra nodes. That is, instead of having a child set to **null**, we can have a special tree node that represents a **dummy** tree and simply have all leaves point to that special tree node. In a sense, then, these dummy tree nodes become the leaves of the tree:



In the above picture, the black circles are **BinaryTree** objects and the black boxes indicate that the values of the left and right children are **null**. So, the example above adds 18 dummy nodes to the tree. The dummy nodes are known as ...

A Sentinel Node (or Sentinel) is a node that represents a path terminator. It is a specifically designated node that is not a data node of the data structure.

Sentinels are used as an alternative over using **null** as the path terminator in order to get one or more of the following benefits: (1) Increased speed of operations; (2) Reduced algorithmic code size; (3) Increased data structure robustness (arguably).

How do we make a sentinel? It is simply a regular **BinaryTree** but has its data value (and children) set to **null** as follows:

```
new BinaryTree(null, null, null)
```

If we decide to use sentinel tree nodes, then we need to add a constructor, perhaps a default constructor and then make changes to the other constructors as necessary to use sentinel nodes instead of **null**.

```
public class BinaryTree2 {
    private String      data;
    private BinaryTree2 leftChild;
    private BinaryTree2 rightChild;

    // A constructor that makes a Sentinel node
    public BinaryTree2() {
        data = null;
        leftChild = null;
        rightChild = null;
    }

    // This constructor now uses sentinels for terminators instead of null
    public BinaryTree2(String d) {
        data = d;
        leftChild = new BinaryTree2();
        rightChild = new BinaryTree2();
    }

    // This constructor is unchanged
    public BinaryTree2(String d, BinaryTree2 left, BinaryTree2 right) {
        data = d;
        leftChild = left;
        rightChild = right;
    }

    // Get methods
    public String getData() { return data; }
    public BinaryTree2 getLeftChild() { return leftChild; }
    public BinaryTree2 getRightChild() { return rightChild; }

    // Set methods
    public void setData(String d) { data = d; }
    public void setLeftChild(BinaryTree2 left) { leftChild = left; }
    public void setRightChild(BinaryTree2 right) { rightChild = right; }
}
```

Now, we can re-write the test case to replace **null** with the new Sentinel nodes:

```
public class BinaryTreeTest2 {
    public static void main(String[] args) {
        BinaryTree2    root;

        root = new BinaryTree2("A",
            new BinaryTree2("B",
                new BinaryTree2("C",
                    new BinaryTree2("D"),
                    new BinaryTree2("E",
                        new BinaryTree2("F",
                            new BinaryTree2("G"),
                            new BinaryTree2("I")),
                        new BinaryTree2("H"))),
                new BinaryTree2("J",
                    new BinaryTree2("K",
                        new BinaryTree2(),
                        new BinaryTree2("L",
                            new BinaryTree2(),
                            new BinaryTree2("M"))),
                    new BinaryTree2("N",
                        new BinaryTree2(),
                        new BinaryTree2("O")))),
            new BinaryTree2("P",
                new BinaryTree2("Q"),
                new BinaryTree2("R",
                    new BinaryTree2("S",
                        new BinaryTree2("T"),
                        new BinaryTree2()),
                    new BinaryTree2("U"))));
    }
}
```

Now we will see the advantage of doing all this. We can re-write the **height()** method so that it does not need to check whether or not the children are **null**, but simply needs to stop the recursion if a sentinel node has been reached:

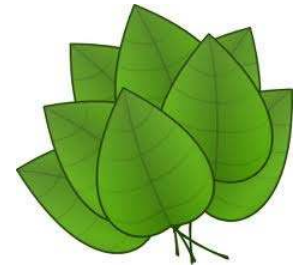
```
public int height() {
    // Check if this is a sentinel node
    if (data == null)
        return -1;

    return 1 + Math.max(leftChild.height(),
                        rightChild.height());
}
```

Notice that since the sentinel nodes have added an extra level to the tree, when we reach a sentinel node, we can indicate a **-1** value as the height so that the path from the leaf to the sentinel does not get counted (i.e., it is essentially subtracted afterwards). The code is MUCH shorter and simpler. This is the advantage of using sentinels ... we do not have to keep checking for **null** values in our code.

Example:

How could we write code that gathers the leaves of a tree and returns them? Again, we will use recursion. In our example, we had 9 leaves. A leaf is identified as having no children, so whenever we find such a node we simply need to add it to a collection.



We can use an **ArrayList<String>** to store the node data. The base case is simple. If the **BinaryTree** is a leaf, return an **ArrayList** with the single piece of data in it:

```
public ArrayList<String> leafData() {
    ArrayList<String> result = new ArrayList<String>();

    if (leftChild == null) {
        if (rightChild == null)
            result.add(data);
    }

    return result;
}
```

Now what about the recursive part? Well, we would have to check both sides of the root as we did before, provided that they are not **null**. Each time, we take the resulting collection of data and merge it with the result that we have so far. The merging can be done using the **list1.addAll(list2)** method in the **ArrayList** class. This method adds all the elements from **list2** to **list1**.

Here is the code:

```
public ArrayList<String> leafData() {
    ArrayList<String> result = new ArrayList<String>();

    if (leftChild == null) {
        if (rightChild == null)
            result.add(data);
        else
            result.addAll(rightChild.leafData());
    }
    else {
        result.addAll(leftChild.leafData());
        if (rightChild != null)
            result.addAll(rightChild.leafData());
    }
    return result;
}
```

What would this code look like with the Sentinel version of our tree? Simpler ...


```

public ArrayList<String> leafData() {
    ArrayList<String> result = new ArrayList<String>();

    if (data != null) {
        if ((leftChild.data == null) && (rightChild.data == null))
            result.add(data);
        result.addAll(leftChild.leafData());
        result.addAll(rightChild.leafData());
    }
    return result;
}

```

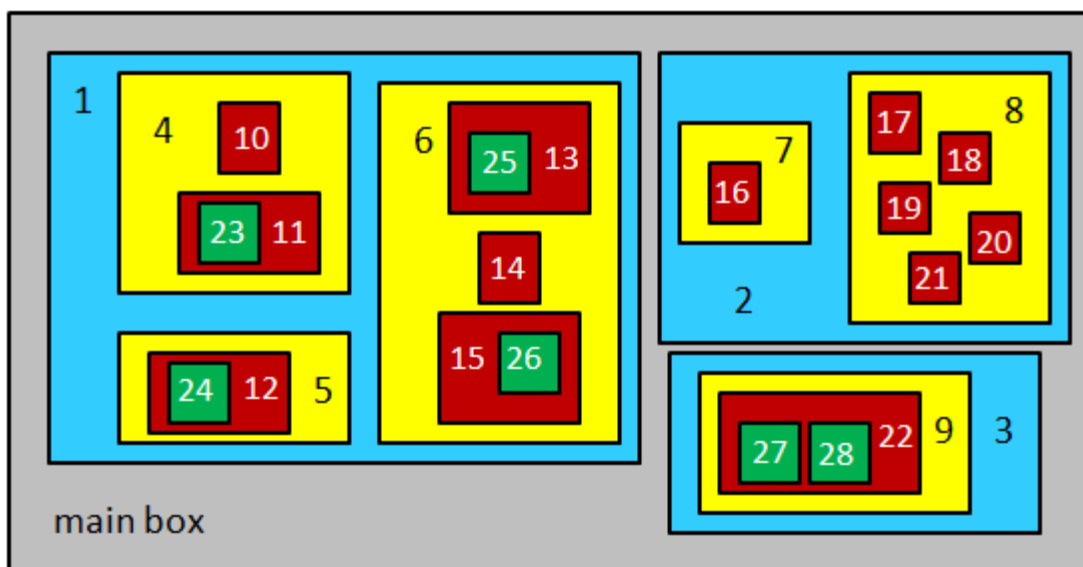
There are many other interesting methods you could write for trees. You will learn more about this next year.

Example:

As another example, consider the following scenario. You wrap up your friends gift in a box ... but to be funny, you decide to wrap that box in a box and that one in yet another box. Also, to fool him/her you throw additional wrapped boxes inside the main box.



This boxes-within-boxes scenario is recursive. So, we have boxes that are completely contained within other boxes and we would like to count how many boxes are completely contained within any given box. Here is an example where the outer (gray) box has 28 internal boxes:



Assume that each box stores an **ArrayList** of the boxes inside of it. We would define a box then as follows:

```

import java.util.ArrayList;

public class Box {
    private ArrayList<Box> internalBoxes;

    // A constructor that makes a box with no boxes in it
    public Box() {
        internalBoxes = new ArrayList<Box>();
    }

    // Get method
    public ArrayList<Box> getInternalBoxes() { return internalBoxes; }

    // Method to add a box to the internal boxes
    public void addBox(Box b) {
        internalBoxes.add(b);
    }

    // Method to remove a box from the internal boxes
    public void removeBox(Box b) {
        internalBoxes.remove(b);
    }
}

```

We could create a box with the internal boxes as shown in our picture above as follows:

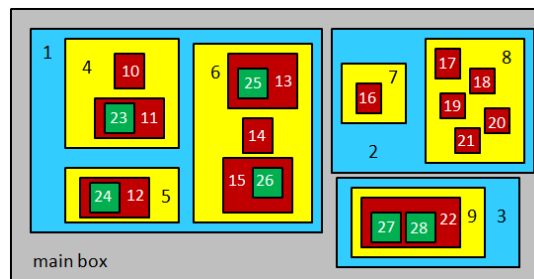
```

public class BoxTest {
    public static void main(String[] args) {
        Box mainBox, a, b, c, d, e, f, g, h, i, j;

        mainBox = new Box();

        // Create the left blue box and its contents
        a = new Box(); // box 10
        b = new Box(); // box 11
        b.addBox(new Box()); // box 23
        c = new Box(); // box 4
        c.addBox(a);
        c.addBox(b);
        d = new Box(); // box 12
        d.addBox(new Box()); // box 24
        e = new Box(); // box 5
        e.addBox(d);
        f = new Box(); // box 13
        f.addBox(new Box()); // box 25
        g = new Box(); // box 14
        h = new Box(); // box 15
        h.addBox(new Box()); // box 26
        i = new Box(); // box 6
        i.addBox(f);
        i.addBox(g);
        i.addBox(h);
        j = new Box(); // box 1
        j.addBox(c);
        j.addBox(e);
        j.addBox(i);
        mainBox.addBox(j);
    }
}

```



```
// Create the top right blue box and its contents
a = new Box(); // box 7
a.addBox(new Box()); // box 16
b = new Box(); // box 8
b.addBox(new Box()); // box 17
b.addBox(new Box()); // box 18
b.addBox(new Box()); // box 19
b.addBox(new Box()); // box 20
b.addBox(new Box()); // box 21
c = new Box(); // box 2
c.addBox(a);
c.addBox(b);
mainBox.addBox(c);

// Create the bottom right blue box and its contents
a = new Box(); // box 22
a.addBox(new Box()); // box 27
a.addBox(new Box()); // box 28
b = new Box(); // box 9
b.addBox(a);
c = new Box(); // box 3
c.addBox(b);
mainBox.addBox(c);
}
}
```

Now, how could we write a function to unwrap a box (as well as all boxes inside of it until there are no more) and return the number of boxes that were unwrapped in total (including the outer box) ?

Do you understand that this problem can be solved recursively, since a **Box** is made up of other **Boxes** ? The problem is solved similarly to the binary tree example since we can view the main box as the "root" of the tree while the boxes inside of it would be considered the children (possibly more than 2).

What is the base case(s) ? What is the simplest box ? Well, a box with no internal boxes would be easy to unwrap and then we are done and there is a total of 1 box:

```
public int unwrap() {
    if (numInternalBoxes == 0)
        return 1;
}
```

This is simple. However, what about the recursive case ? Well, we would have to recursively unwrap and count all inside boxes. So we could use a loop to go through the internal boxes, recursively unwrapping them one-by-one and totaling the result of each recursive unwrap call as follows:

```

public int unwrap() {
    if (internalBoxes.size() == 0)
        return 1;
    // Count this box
    int count = 1;

    // Count each of the inner boxes
    for (Box b: internalBoxes)
        count = count + b.unwrap();
    return count;
}

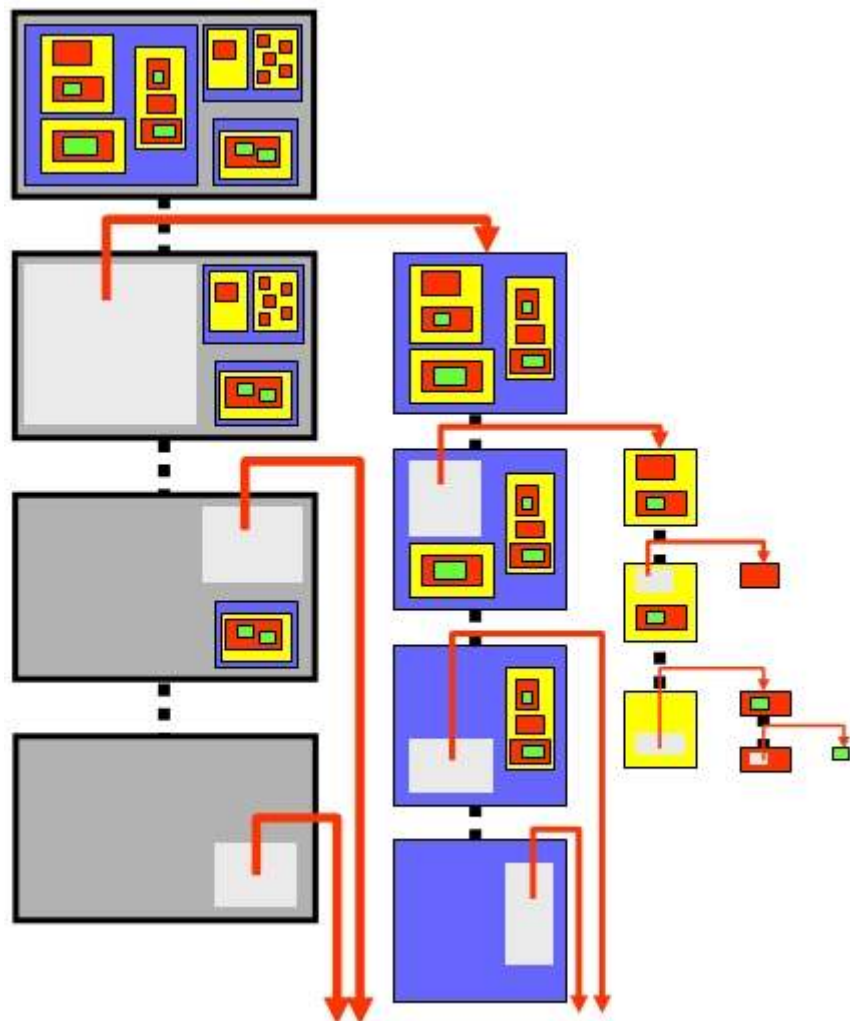
```

Notice how we adjusted the base case. The **for** loop will attempt to unwrap all internal boxes recursively. If there are no internal boxes, then the method returns a value of 1 ... indicating this single box.

Each recursively-unwrapped box has a corresponding **count** representing the number of boxes that were inside of it (including itself). These are all added together to obtain the result.

The above code does not modify the **internalBoxes** list for any of the boxes. That is, after the function has completed, the box data structure remains intact and unmodified. This is known as a **non-destructive** solution because it does not destroy (or alter) the data structure. In real life however, the boxes are actually physically opened and the contents of each box is altered so that when completed, no box is contained in any other boxes (i.e., the list is modified/destroyed).

Alternatively, we can obtain the same solution without a **for** loop by allowing the arrayLists to be destroyed along the way. This would be called a **destructive** solution. Destructive solutions are often simpler to code and understand, but they have the disadvantage of a modified data structure, which can be undesirable in some situations. Here is the process depicting a portion of such a destructive solution: →



How does this simplify our code ? If we are not worried about keeping the **innerBoxes** lists intact, we can simply "bite-off" a piece of our problem by removing one box from the main box (i.e., taking it out of the list of internal boxes) and then we have two smaller problems:

- (1) the original box with one less internal box in it, and
- (2) the box that we took out that still needs to be unwrapped.

We can simply unwrap each of these recursively and add their totals together:

```
public int unwrap2() {
    if (internalBoxes.size() == 0)
        return 1;
    // Remove one internal box, if there is one
    Box insideBox = internalBoxes.remove(0);

    // Unwrap the rest of this box as well as the one just removed
    return this.unwrap2() + insideBox.unwrap2();
}
```

This is much smaller code now. It is also intuitive when you make the connection to the real-life strategy for unwrapping the boxes.

Of course, once the method completes ... the main box is empty ... since the boxes were removed from the array list along the way. If we wanted to ensure that the main box remained the same, we could put back the boxes after we counted them. This would have to be done AFTER the recursive calls ... but before the **return** statement:

```
public int unwrap3() {
    if (internalBoxes.size() == 0)
        return 1;
    // Remove one internal box, if there is one
    Box insideBox = internalBoxes.remove(0);

    // Unwrap the rest of this box as well as the one just removed
    int result = this.unwrap3() + insideBox.unwrap3();

    // Put the box back in at position 0 (i.e., same order)
    internalBoxes.add(0, insideBox);

    return result;
}
```

This method is now considered **non-destructive** because the boxes are restored before the method completes.

9.3 A Maze Searching Example

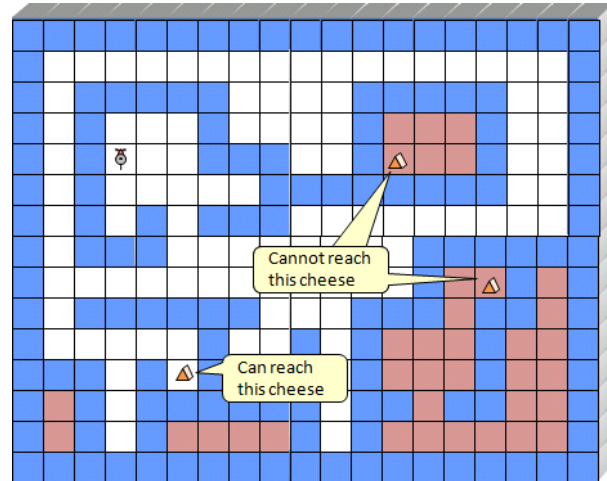
Consider a program in which a rat follows the walls of a maze. The rat is able to travel repeatedly around the maze using the “right-hand rule”. Some mazes may have areas that are unreachable (e.g., interior rooms of a building with closed doors). We would like to write a program that determines whether or not a rat can reach a piece of cheese that is somewhere else in the maze.

This problem cannot be solved by simply checking each maze location one time. It is necessary to trace out the steps of the rat to determine whether or not there exists a path from the rat to the cheese.

To do this, we need to allow the rat to try all possible paths by propagating (i.e., spreading) outwards from its location in a manner similar to that of a fire spreading outwards from a single starting location.

Unlike a fire spreading scenario, we do not have to process the locations in order of their distance from the rat’s start location. Instead, we can simply allow the rat to keep walking in some direction until it has to turn, and then choose which way to turn. When there are no more places to turn to (e.g., a dead end), then we can return to a previous “crossroad” in the maze and try a different path.

We will need to define a **Maze** class that the **Rat** can move in. Here is a basic class that allows methods for creating and displaying a maze as well as adding, removing and querying walls:



```
public class Maze {
    public static byte EMPTY = 0;
    public static byte WALL = 1;
    public static byte CHEESE = 2;

    private int rows, columns;
    private byte[][] grid;

    // A constructor that makes a maze of the given size
    public Maze(int r, int c) {
        rows = r;
        columns = c;
        grid = new byte[r][c];
    }
    // A constructor that makes a maze with the given byte array
    public Maze(byte[][] g) {
        rows = g.length;
        columns = g[0].length;
        grid = g;
    }
}
```

```

// Return true if a wall is at the given location, otherwise false
public boolean wallAt(int r, int c) { return grid[r][c] == WALL; }

// Return true if a cheese is at the given location, otherwise false
public boolean cheeseAt(int r, int c) { return grid[r][c] == CHEESE; }

// Put a wall at the given location
public void placeWallAt(int r, int c) { grid[r][c] = WALL; }

// Remove a wall from the given location
public void removeWallAt(int r, int c) { grid[r][c] = EMPTY; }

// Put cheese at the given location
public void placeCheeseAt(int r, int c) { grid[r][c] = CHEESE; }

// Remove a cheese from the given location
public void removeCheeseAt(int r, int c) { grid[r][c] = EMPTY; }

// Display the maze in a format like this ----->
public void display() {
    for(int r=0; r<rows; r++) {
        for (int c = 0; c<columns; c++) {
            if (grid[r][c] == WALL)
                System.out.print("W");
            else if (grid[r][c] == CHEESE)
                System.out.print("c");
            else
                System.out.print(" ");
        }
        System.out.println();
    }
}

// Return a sample maze corresponding to the one in the notes
public static Maze sampleMaze() {
    byte[][] grid = {
        {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
        {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
        {1,0,1,1,1,1,1,0,0,0,0,1,1,1,1,1,0,0,1},
        {1,0,1,0,0,0,1,0,0,0,0,1,0,0,0,1,0,0,1},
        {1,0,1,0,0,0,1,1,1,0,0,1,0,0,0,1,0,0,1},
        {1,0,1,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,1},
        {1,0,1,0,1,0,1,1,1,0,0,0,0,0,0,0,0,0,1},
        {1,0,1,1,1,0,0,0,0,0,0,0,0,0,1,1,1,1,1},
        {1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,1},
        {1,0,1,1,1,1,1,1,0,0,0,1,1,1,0,1,1,0,1},
        {1,0,0,0,0,0,1,0,0,1,0,1,0,0,0,1,0,0,1},
        {1,1,1,0,1,0,1,0,0,1,0,1,0,0,0,1,0,0,1},
        {1,0,1,0,1,1,1,1,1,1,0,1,1,0,1,1,0,0,1},
        {1,0,1,0,1,0,0,0,0,0,1,0,1,0,0,0,0,0,1},
        {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}};

    Maze m = new Maze(grid);
    m.placeCheeseAt(3,12);
    return m;
}
}

```

```

WWWWWWWWWWWWWWWWWW
W          W
W WWWWW   WWWWW W
W W  W   Wc  W W
W W  WWW  W  W W
W W          WWWWWW W
W W W WWW          W
W WWW          WWWWWW
W          W  W W
W WWWWWW   WWW WW W
W  W  W W  W  W W
WWW W W W W  W W
W W WWWWWW WW WW W
W W W  W W  W
WWWWWWWWWWWWWWWWWW

```

What does the **Rat** class look like? It is very simple, for now:

```

public class Rat {
    private int row, col;

    // Move the Rat to the given position
    public void moveTo(int r, int c) {
        row = r; col = c;
    }
}

```

Let us see whether or not we can write a recursive function in the **Rat** class to solve this problem. The function should take as parameters the maze (i.e., a 2D array). It should return **true** or **false** indicating whether or not the cheese is reachable from the rat's location. Consider the method written in a **Rat** class as follows:

```

public boolean canFindCheeseIn(Maze m) {
    ...
}

```

What are the base cases for this problem? What is the simplest scenario? To make the code simpler, we will assume that the entire maze is enclosed with walls ... that is ... the first row, last row, first column and last column of the maze are completely filled with walls.

There are two simple cases:

1. If the cheese location is the same as the rat's location, we are done ... the answer is true.
2. If the rat is on a wall, then it cannot move, so the cheese is not reachable. This is a kind of error-check, but as you will see later, it will simplify the code.

Here is the code so far:

```

public boolean canFindCheeseIn(Maze m) {
    // Return true if there is cheese at the rat's (row,col) in the maze
    if (m.cheeseAt(row, col))
        return true;

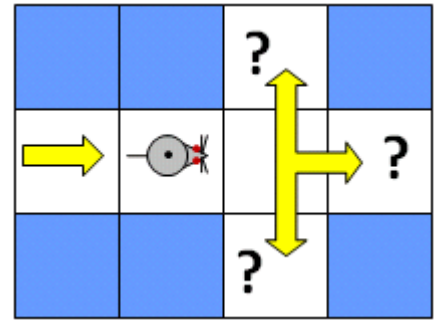
    // Return false if there is a wall at the rat's (row,col) in the maze
    if (m.wallAt(row, col))
        return false;
}

```

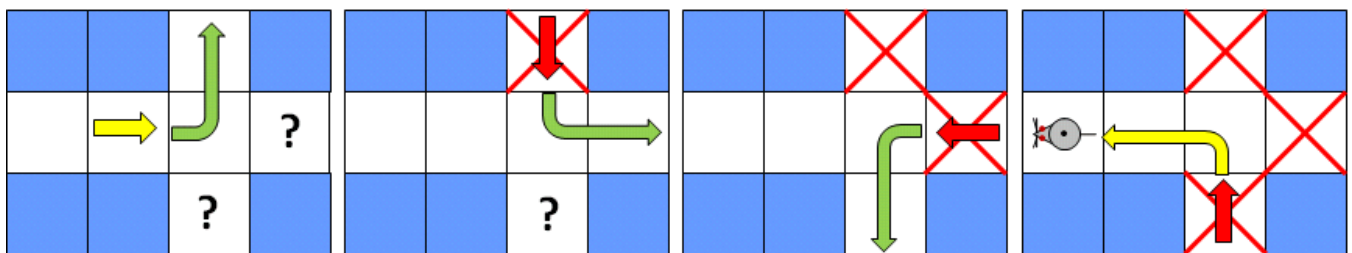
Notice that the **row** and **col** variables are the attributes of the **Rat** itself.

Now what about the recursion? How do we "break off" a piece of the problem so that the problem becomes smaller and remains the same type of problem? Well, how would you solve the problem if you were the rat looking for the cheese?

Likely, you would start walking in some direction looking for the cheese. If there was only one single path in the maze you would simply follow it. But what do you do when you come to a crossroads (i.e., a location where you have to make a decision as to which way to go) ?



Probably, you will choose one of these unexplored hallways, and then if you find the cheese ... great. If you don't find the cheese down that hallway, you will likely come back to that spot and try a different hallway. If you find the cheese down any one of the hallways, your answer is **true**, otherwise...if all hallways "came up empty" with no cheese, then you have exhausted all possible routes and you must return **false** as an answer for this portion of the maze:



So the idea of "breaking off" a smaller piece of the problem is the same idea as "ruling out" one of the hallways as being a possible candidate for containing the cheese. That is, each time we check down a hallway for the cheese and come back, we have reduced the remaining maze locations that need to be searched.

This notion can be simplified even further through the realization that each time we take a step to the next location in the maze, we are actually reducing the problem since we will have already checked that location for the cheese and do not need to re-check it. That is, we can view each location around the rat as a kind of hallway that needs to be checked. So, the general idea for the recursive case is as follows:

```

if (the cheese is found on the path to the left) then return true
otherwise if (the cheese is found on the path straight ahead) then return true
otherwise if (the cheese is found on the path to the right) then return true
otherwise return false

```

There are only three possible cases, since we do not need to check behind the rat since we just came from that location. However, the actual code is a little more complicated. We need, for example, to determine the locations on the "left", "ahead" and "right" of the rat, but this depends on which way the rat is facing. There would be the three cases for each of the 4 possible rat-facing directions. A simpler strategy would simply be to check all 4 locations around the rat's current location, even though the rat just came from one of those locations. That way, we can simply check the 4 maze locations in the array around the rat's current location.

Here is the idea behind the recursive portion of the code:

```

move the rat up
if (canFindCheese(maze)) then return true otherwise move the rat back down

move the rat down
if (canFindCheese(maze)) then return true otherwise move the rat back up

move the rat left
if (canFindCheese(maze)) then return true otherwise move the rat back right

move the rat right
if (canFindCheese(maze)) then return true otherwise move the rat back left

return false

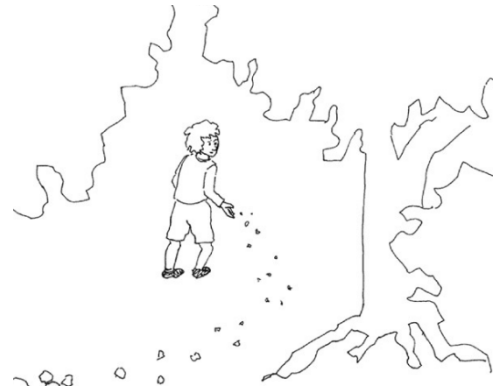
```

The code above has 4 recursive calls. It is possible that all 4 recursive calls are made and that none of them results in the cheese being found.

However, there is a problem in the above code. With the above code, the rat will walk back and forth over the same locations many times ... in fact ... the code will run forever ... it will not stop. The problem is that each time we call the function recursively, we are not reducing the problem. In fact, each time, we are simply starting a brand new search from a different location.

The rat needs a way of “remembering” where it has been before so that it does not “walk in circles” and continue checking the same maze locations over and over again. To do this, we need to leave a kind of “breadcrumb trail” so that we can identify locations that have already been visited.

We can leave a “breadcrumb” at a maze location by changing the value in the array at that row and column with a non-zero & non-wall value such as **-1**. Then, we can treat all **-1** values as if they are walls by not going over those locations again.



We can add code to the **Maze** class to do this:

```

public static byte BREAD_CRUMB = -1;

// Mark the given location as visited
public void markVisited(int r, int c) {
    grid[r][c] = BREAD_CRUMB;
}

// Mark the given location as not having been visited
public void markUnVisited(int r, int c) {
    grid[r][c] = EMPTY;
}

// Return true if the location has been visited
public boolean hasBeenVisited(int r, int c) {
    return grid[r][c] == BREAD_CRUMB;
}

```

Now we can adjust our code to avoid going to any "visited locations" and to ensure that each location is visited. We can also put in the code to do the recursive checks now as follows:

```
public boolean canFindCheeseIn(Maze m) {
    // Return true if there is cheese at the rat's (row,col) in the maze
    if (m.cheeseAt(row, col))
        return true;

    // Return false if there is a wall at the rat's (row,col) in the maze
    if (m.wallAt(row, col) || m.hasBeenVisited(row, col))
        return false;

    // Mark this location as having been visited
    m.markVisited(row, col);

    // Move up in the maze and recursively check
    moveTo(row-1, col);
    if (canFindCheeseIn(m))
        return true;

    // Move back down and then below in the maze and recursively check
    moveTo(row+2, col);
    if (canFindCheeseIn(m)) return true;

    // Move back up and then left in the maze and recursively check
    moveTo(row-1, col-1);
    if (canFindCheeseIn(m)) return true;

    // Move back and then go right again in the maze and recursively check
    moveTo(row, col+2);
    if (canFindCheeseIn(m)) return true;

    // We tried all directions and did not find the cheese, so quit
    return false;
}
```

Notice that we are now returning with **false** if the location that the rat is at is a wall or if it is a location that has already been travelled on. Also, we are setting the rat's current maze location to **-1** so that we do not end up coming back there again.

After running this algorithm, the maze will contain many **-1** values. If we wanted to use the same maze and check for a different cheese location, we will need to go through the maze and replace all the **-1** values with **0** so that we can re-run the code. This recursive function is therefore considered to be destructive. Destructive functions are not always desirable since they affect the outcome of successive function calls.

However, there is a way to fix this right in the code itself. Notice that we are setting the maze location to **-1** just before the recursive calls. This is crucial for the algorithm to work. However, once the recursive calls have completed, we can simply restore the value to **0** again by placing the following just before each of the **return** calls:

```
m.markUnVisited(row, col);
```

For example:

```
// Move up in the maze and recursively check
moveTo(row-1, col);
if (canFindCheeseIn(m)) {
    moveTo(row+1, col);           // Move back down before marking
    m.markUnvisited(row, col);   // Unmark the visited location
    return true;
}
```

The code above should now do what we want it to do.

We need to test the code. To help debug the code it would be good to be able to display where the rat is and where the breadcrumbs are.

We can modify the **display()** method in the **Maze** class to take in the rat's location and do this as follows:

```
public void display(int ratRow, int ratCol) {
    for(int r=0; r<rows; r++) {
        for (int c = 0; c<columns; c++) {
            if ((r == ratRow) && (c == ratCol))
                System.out.print("r");
            else if (grid[r][c] == WALL)
                System.out.print("W");
            else if (grid[r][c] == CHEESE)
                System.out.print("c");
            else if (grid[r][c] == BREAD_CRUMB)
                System.out.print(".");
            else
                System.out.print(" ");
        }
        System.out.println();
    }
}
```

We can then use the following test program:

```
public class MazeTest {
    public static void main(String[] args) {
        Maze m = Maze.sampleMaze();
        Rat r = new Rat();
        r.moveTo(1,1);
        m.display(1,1);
        System.out.println("Can find cheese ... " +
            r.canFindCheeseIn(m));
    }
}
```

Also, by inserting **m.display(row,col);** at the top of your recursive method, you can watch as the rat moves through the maze:

This page was intentionally left blank.

Chapter 10

Exception Handling

What is in This Chapter ?

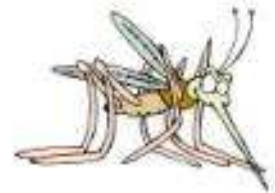
It is never possible to predict accurately what the user of your software will do. While your program is running, situations often arise in which some unexpected error occurs, perhaps due to unexpected or corrupt data. We have to deal with these problems gracefully in our code so that our code is robust, produces valid/correct results and does not crash. In this set of notes, we will discuss **Exceptions**, which are JAVA's way of handling problems that occur in your program. You will find out how to handle standard problems that occur in your code by using the **Exception** classes and how to define your own types of **Exceptions**.



10.1 Simple Debugging

We use the term **bug** in computer science to denote a problem with our program. Unfortunately, much of our programming time may be spent on finding errors/bugs in the code that we write. This can be VERY time consuming and frustrating. Sometimes we may fix one bug only to find that another one appears. There are basically 3 types of errors (i.e., bugs):

- 1. Compile Errors** occur when your code will not compile. They are the easiest to find since the compiler catches them and informs us of the problem. Because JAVA is strongly typed, many “misunderstandings” between method parameters and variables are eliminated. Once fixed, compiler errors do not come back. Often though ... one error (such as a missing semicolon) can lead to a whole slew of compile errors.
- 2. Runtime Errors** cannot be determined at compile time. They “pop up” when you run your code and usually represent a serious problem (e.g., divide by zero, stack overflow, out of memory). These errors may sometimes require a re-design of your code (e.g., to reduce memory usage). But often, the problem is less serious such as trying to send messages to a **null** object (i.e., **NullPointerException**) or accessing past available array boundaries (i.e., **ArrayOutOfBoundsException**)
- 3. Logic Errors** pertain to the logistics of your program such as computing wrong values or forgetting to handle certain “special situations” in your code. JAVA **cannot detect** nor **explain** these errors. Sometimes the logic error could lead to a runtime error which JAVA can then catch, but it certainly cannot explain them. Logic errors are often VERY difficult to find since the program could “appear” to be working. Rigorous testing is required to find them. Logic errors typically require you to do some *debugging*.



As programmers, we spend much of our time maintaining code and doing what is known as ...

Debugging is the process of “figuring out errors” in your program and “fixing” them.



Actually, finding the error is usually the hard part. Fixing it is often (but certainly not always) easy. One of the most common debugging techniques is that of using “print” statements in your code. When there are many logic errors, this is usually the simplest way to debug.



If your program is producing wrong answers, you can use **print** statements to display intermediate calculations as follows ...


```
public double computeMortgagePayment() {
    double monthlyRate = this.getInterestRate() / 12;
    System.out.println("monthly rate = " + monthlyRate);    // debug

    double amortizeRate = (1-Math.pow(1+monthlyRate, this.numMonths*-1));
    System.out.println("amortize rate = " + amortizeRate); // debug
    return this.getHousePrice() * monthlyRate / amortizeRate;
}
```

From the intermediate results, you should be able to narrow down where you went wrong. Print statements can also be used to determine whether or not a certain point in your code is being reached or if a certain method is being called. (this is especially useful when programming in C):

```
public double computeMortgagePayment() {
    System.out.println("*** Got Here 1");
    double monthlyRate = this.getInterestRate() / 12;
    double amortizeRate = (1-Math.pow(1+monthlyRate, this.numMonths*-1));

    System.out.println("*** Got Here 2");

    return this.getHousePrice() * monthlyRate / amortizeRate;
}
```

By doing this, we can get an idea as to where our program has stopped working and also find out if JAVA is calling the methods that we think it is calling. Print statements can also be used to show the order that certain pieces of code are evaluated in: (this is especially useful when using timer events or when multiple processes are running)

```
public void deposit(float anAmount) {
    System.out.println("depositing $" + anAmount);
    this.balance = this.balance + anAmount;
}

public boolean withdraw(float anAmount) {
    System.out.println("withdrawing $" + anAmount);
    if (anAmount <= this.balance) {
        this.balance = this.balance - anAmount;
        return true;
    }
    return false;
}
```

In order to simplify the print statements, we can often print out whole objects ...

```
public static void Test1() {
    BankAccount account = new BankAccount("Jim");
    account.deposit(120.53f);
    account.withdraw(20);
    account.deposit(400);
    account.withdraw(829.31f);
    System.out.println(account);
}
```

As long as we have implemented an informative **toString()** method for our objects, we should get descriptive output.

Although this debugging technique is effective, your code may become littered with **System.out.println** statements which need to eventually be removed before you ship out your code. However, the "print statement" remains one of the most popular and simplest methods for debugging and this technique will usually help us narrow down the error that occurred.

In JAVA, it seems that the most common errors occur because we *forgot to initialize* something or if *unexpected* data was given to us. In some cases we can write additional code to "expect and handle" bad input data. This is called **error-checking** and it is the basis for **Exceptions** in JAVA. We will not discuss debugging any further in this course, but will instead focus on how to deal gracefully with unexpected errors that may arise in our programs.

10.2 Exceptions

There are many chances for errors to occur in a program when the programmer has no control over information that is entered into the program from the keyboard, files, or from other methods/classes/packages etc... Even worse ... when such errors occur, it is not always clear how to handle the error.



Exceptions are errors that occur in your program.

They are JAVA's way of telling you that something has gone wrong in your program. When an exception occurs, JAVA forces us to do one of the following:

1. Handle the exception (we must know **when** to do this and **what** to do), or
2. Declare that we want **someone else** to handle it.

Exception Handling is the strategy of handling errors which are generated during program execution

We handle exceptions in order to allow our program to "quit gracefully" as opposed to having JAVA spew out a bunch of exception messages.

When should we handle exceptions? When we do not know how to deal with the error ... or when it does not make sense to handle the error.

For example, in large software systems, an error may occur outside of the code that we wrote (i.e., in someone else's code). We may not even have access to this code in order to fix the error. Perhaps the error occurred in some module that was developed by another team of programmers. Sometimes, it is an advantage to anticipate some possible errors and then we can allow our program to handle the error gracefully. However, it is sometimes the case that we do not know what to do at all when the error occurs. If our code can easily predict a particular kind of error, then there is no need to use **Exceptions**, since we can deal with the code on our own.

Furthermore, in software components such as methods, libraries, and classes that are likely to be widely used, it is unclear as to what should be done when the error occurs. Our decision as to how we handle the error may or may not be the best choice for the software as a whole.

To help you understand, consider this "real world" example in which an unexpected situation occurs. Suppose that you ask your friend to go to McDonald's to get you a Big Mac and Fries. You expect him to come back with food for you.

However, what could go wrong ?

1. he crashes his car and never arrives at McDonald's
2. he gets there, but the place is burnt down
3. he places his order but finds out there are no Big Macs left anymore
4. he places the order but does not have enough money
5. he gets the food and drops/spills it on the ground on the way back



As you can see, much can go wrong ... but what would your friend do in each of these situations ?

1. he informs you that he cannot handle your request
2. he either returns informing you of the problem, or drives to a different McDonald's or nearby restaurant
3. he improvises and gets you two single hamburgers in the place of the Big Mac
4. he gets you an incomplete order
5. he tries to save money and simply wipes it off ;) ... or perhaps purchases replacements.

As you may well understand by now, we need to think along these lines. We need to always ask ourselves:

- What can go wrong ?
- Should I handle it ?
- How do I handle it ?

As it turns out, JAVA has a nice mechanism for handling errors in a consistent manner. We don't always need to use this mechanism in our code, but there are advantages:

- **Improves clarity** of programs for large pieces of software
- Can be **more efficient** than "home-made" error checking code

- They apply to **multi-threaded** (more than one program) applications
- Programmers **save time** by using predefined Exceptions

In JAVA, exceptions are **thrown** (i.e., generated) by either:

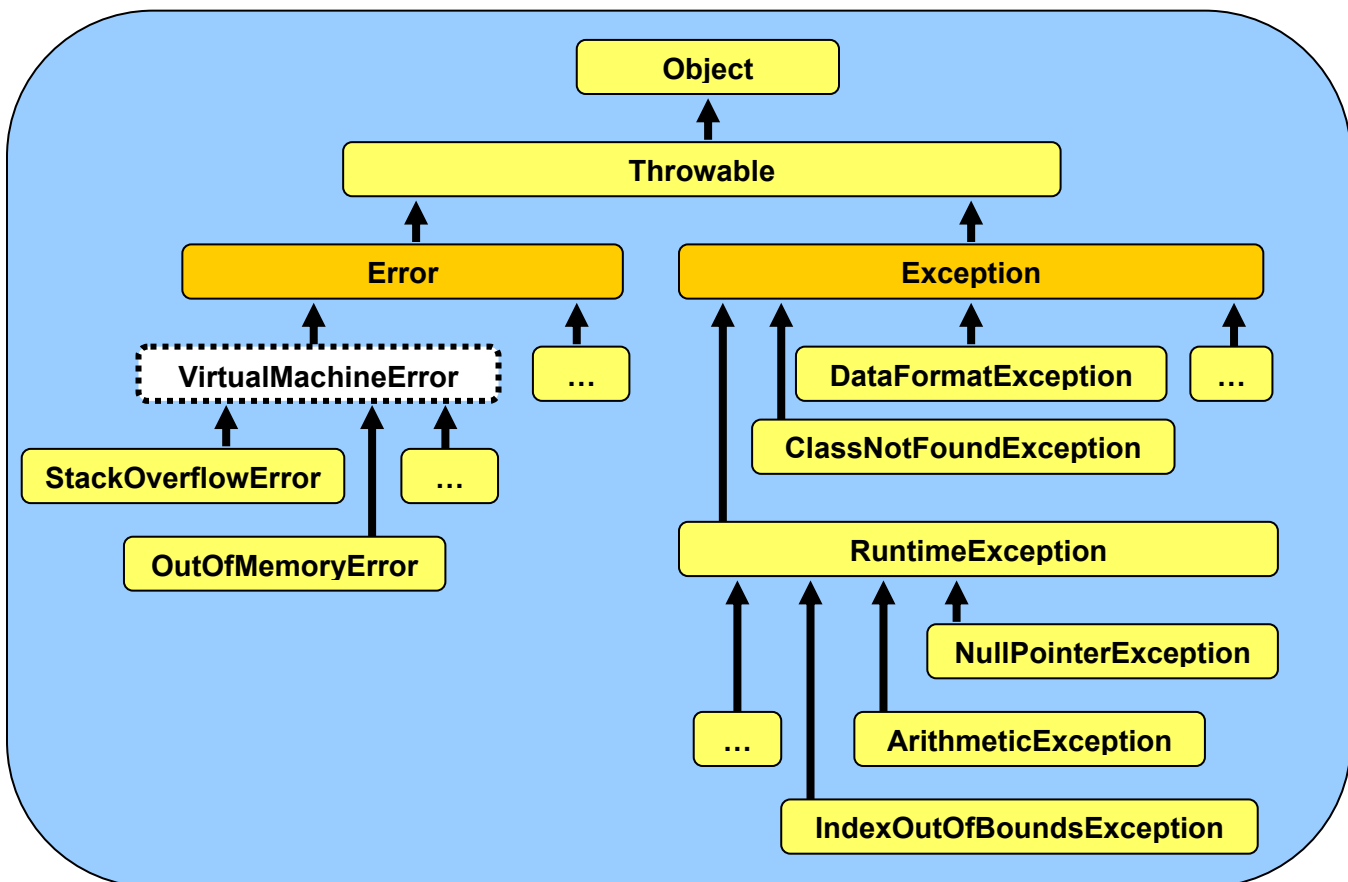
- the JVM automatically
- your code at any time



Exceptions are always **caught** (i.e., handled) by one of these:

- your own code (i.e., graceful decision)
- someone else's code (i.e., delegate the responsibility)
- the JVM (i.e., program halts)

JAVA has many predefined exceptions, and we can also create our own. In JAVA, **Exceptions** are objects, so each one is defined in its own class. The **Exception** classes are arranged in a hierarchy, and their position in the hierarchy can affect the way that they are handled. There are also **Error** objects in JAVA ... which represent more serious errors that may occur in your program which would require the program to stop altogether since they are considered unrecoverable:



In regards to the **Error** classes, generally your application should not try to catch them. There are many subclasses of **Error**, here are just a few:

- `VirtualMachineError`
 - `StackOverflowError` (e.g., recursion too deep)
 - `OutOfMemoryError` (e.g., can't create any more objects)
- `LinkageError`
 - `NoClassDefFoundError` (e.g., no class with given name)
 - `ClassFormatError` (e.g., class is incompatible)



The **Exception** class and its subclasses indicate a "less serious" problem. The exceptions are either "checked" or "unchecked" by the compiler. "Checked" exceptions are pre-defined types of errors that the JAVA compiler looks for in your code and forces you to deal with them before it will compile your code. Generally, your applications will need to deal with these types of **Exceptions**.

Here are just a few of the "checked" exceptions that we might need to catch in our code:

- `ClassNotFoundException` (e.g., tried to load an undefined class)
- `CloneNotSupportedException` (e.g., cannot make copy of object)
- `DataFormatException` (e.g., bad data conversion)
- `IllegalAccessException` (e.g., access modifiers prevent access)
- `InstantiationException` (e.g., problem creating an object)
- `IOException`
 - `EOFException` (e.g., end of file exception)
 - `FileNotFoundException` (e.g., cannot find a specified file)

Here are a few of the "unchecked" exceptions. Although you can check for (i.e., detect and handle) these types of errors in your code, normally you will not do so. Instead, you will try to write your code so that such exceptions cannot happen. The JAVA compiler will not force you to handle these errors before compiling:

- `RuntimeException`
 - `ArithmeticException` (e.g., bad computation such as divide by 0)
 - `ArrayStoreException` (e.g., storing wrong type of object in array)
 - `ClassCastException` (e.g., cannot typecast one class to another)
 - `IndexOutOfBoundsException` (e.g., gone outside array bounds)
 - `NoSuchElementException` (e.g., cannot find any more elements)
 - `NullPointerException` (e.g., attempt to send message to **null**)
 - `NumberFormatException` (e.g., trouble converting to a number)

Recall that when **exception handling** you must either (a) handle the exception yourself, or (b) declare that someone else will handle it.

In the 2nd case, we are actually **delegating** the exception-handling responsibility to someone else. We do this when we do not want to handle the error situation in our code. We actually delegate the responsibility to the "calling method" (i.e., the method that called our method must handle the error). We do this by adding a *throws clause* to our method declaration as follows ...



```
public void openFile(String fileName) throws java.io.FileNotFoundException {
    // code for method
}
```

The **throws** keyword appears at the end of a method signature and is followed by an **Exception** type. When compiling this method, JAVA will check all methods that call this **openFile()** method to make sure that they deal with the **FileNotFoundException** in some way (i.e., either by catching it, or declaring that they too will throw it, thereby delegating the responsibility further up the chain of method calls).

You can actually specify multiple exception types with the **throws** clause by listing the exceptions separated by commas:

```
void convertFile(String fileName) throws java.io.FileNotFoundException,
                                       java.lang.ClassNotFoundException,
                                       java.io.IOException {
    // code for method
}
```

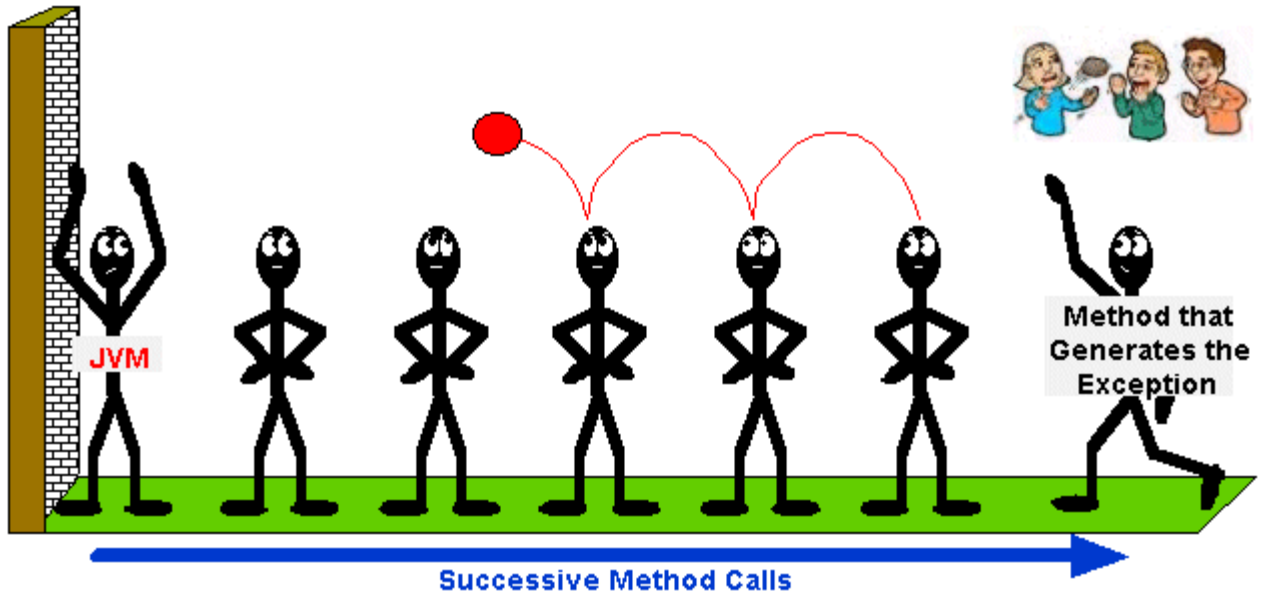
So, to clarify things a little, the **throws** clause is part of a method's declaration that is used to tell the compiler which exceptions the method may throw back to its caller. The **throws** clause is **required** if the code in the method "may" generate, but not handle, a particular type of exception. You should think of the **throws** clause as a "**sign**" that the method holds up in order to tell the whole world publicly that the code in that method may generate the specified exception.



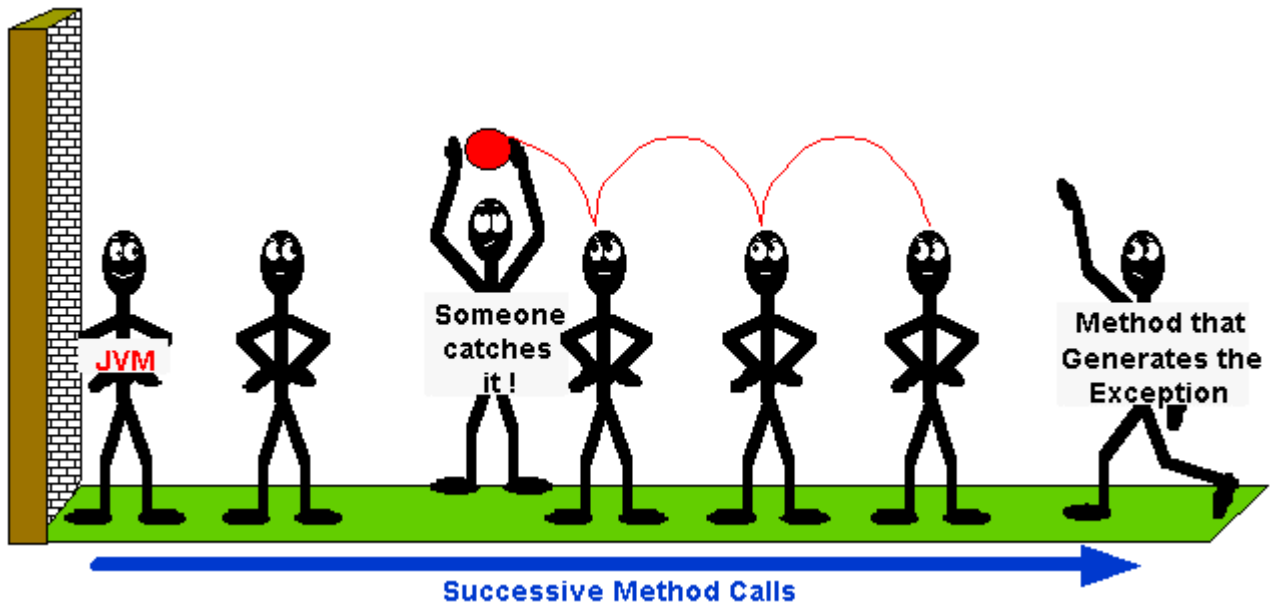
For example, if we consider the **openFile()** method mentioned earlier, it declares to everyone in its signature that it may generate a **FileNotFoundException** at any time. So, when we call the **openFile()** method from some other method, say **getCustomerInfo()**, then the **getCustomerInfo()** method "may also" declare that it throws the exception (if, for example, it did not want to handle it):

```
public void getCustomerInfo() throws java.io.FileNotFoundException {
    // do something
    this.openFile("customer.txt");
    // do something
}
```

Here, if the exception is thrown while in the **openFile()** method, the **getCustomerInfo()** method will stop and it will then pass on the exception to "its" caller. The responsibility may be repeatedly delegated in this manner. It is as if everyone ignores the error (like a hot potato). Nobody explicitly handles the error. The JVM will eventually catch it and halt the program:



At any time during this process however, any method may catch the exception and handle it. Once caught, propagation of the exception stops.



A method may catch an exception by specifying **try** and **catch** blocks. A "block" here refers to a sequence of JAVA statements (i.e., code defined between braces { }).

The "**try** block" represents the code for which you want to handle an Exception. We precede this block with the **try** keyword. Similarly, the "**catch** block" represents the code that handles a particular type of exception. We precede these blocks with the **catch** keyword.

A **catch** block always appears right after a **try** block as follows:

```
...
try {
    // some code that may cause an exception
}
catch (FileNotFoundException ex) {
    // some code that handles the exception
}
...
```

Notice that the **catch** block requires a parameter which indicates the type of error to be caught. This parameter can be accessed and used within the **catch** block (more on this later). The **getCustomerInfo()** method in our previous example can decide to handle the exception through use of **try/catch** blocks as follows:

```
public void getCustomerInfo() {
    try {
        this.openFile("customer.txt");
    }
    catch (java.io.FileNotFoundException ex) {
        System.out.println("Error: File not found"); // Handle the error here
    }
}
```

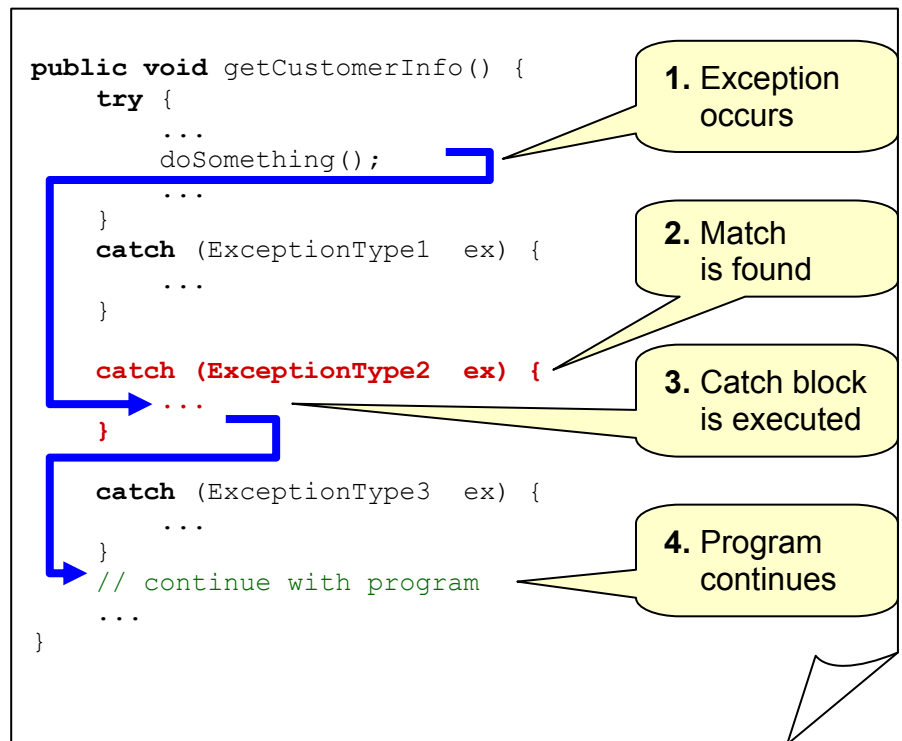
Notice that the method no longer needs to "throw" the exception any further (i.e., no **throws** clause), since it caught and handled it.

More than one **catch** block may be used to catch one-of-many possible exceptions. We simply list all **catch** blocks one after another:

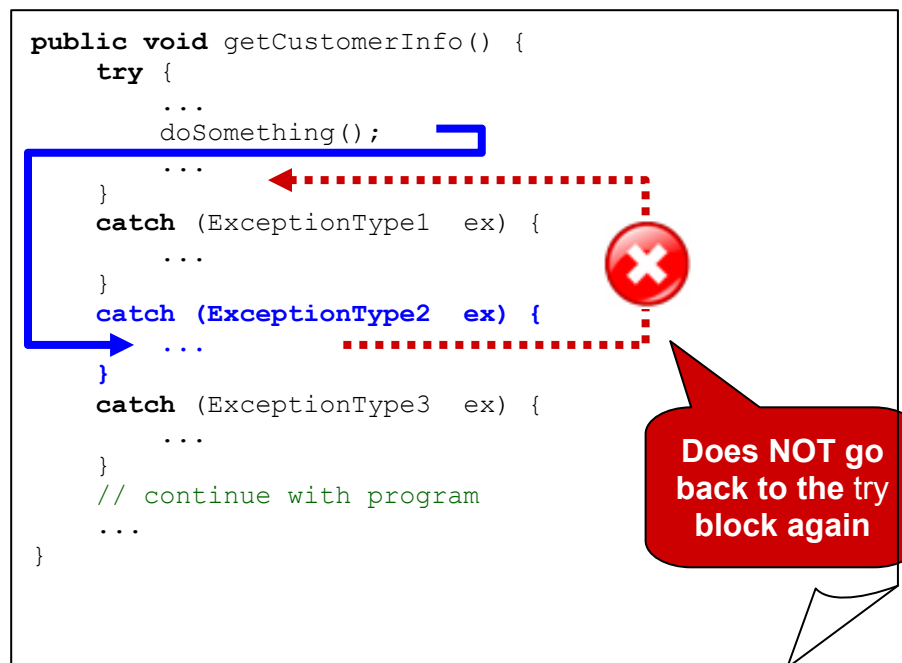
```
public void getCustomerInfo() {
    try {
        // do something that may cause an Exception
    }
    catch (java.io.FileNotFoundException ex) {
        // Handle the error here
    }
    catch (NullPointerException ex) {
        // Handle the error here
    }
    catch (ArithmeticException ex) {
        // Handle the error here
    }
}
```

Consider what happens when an exception occurs within a **try** block:

Here, an exception of type **ExceptionType2** occurs as a result of the **doSomething()** method call. JAVA will immediately stop running the code in the **try** block and search through the **catch** blocks for one whose parameter type matches the exception that occurred (i.e., for one that takes **ExceptionType2** parameter or a superclass of **ExceptionType2**). When one is found, it executes the code within that **catch** block and then continues to the point in the program immediately following the **catch** blocks.



Note that JAVA does NOT go back to the **try** block once it completes the **catch** block. So any code remaining in the **try** block after the location where the exception had occurred is not evaluated as shown here →

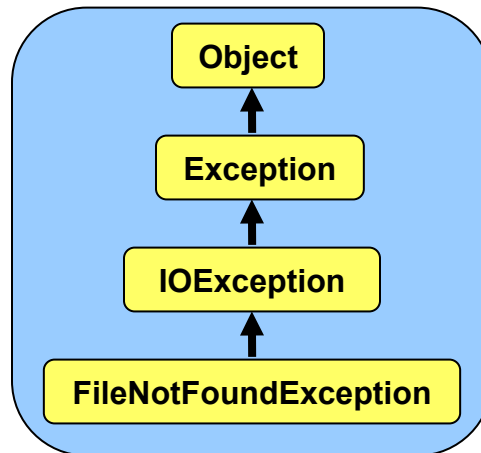


If no match is found when JAVA looks for a matching **catch** block, then the entire **getCustomerInfo()** method halts and the method throws the same exception to the method that called this **getCustomerInfo()**

method and that method will then have to deal with the exception in some way.

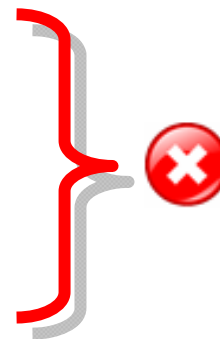
Since **Exceptions** are objects and are organized in a class hierarchy, then one **Exception** may be a more specific kind of another one. That is, **Exceptions** in general may have superclasses and subclasses. So, when JAVA goes looking through the **catch** blocks for a match, it will look for the first match that either matches the **Exception** class exactly or matches one of its superclasses. It is important to note that only one **catch** block (the one that "first" matches the exception) will ever be evaluated. That means we need to be careful, because the order of the **catch** blocks is important when we list them.

Consider the following portion of the JAVA class hierarchy:



The code below is problematic. Do you know why ?

```
public void getCustomerInfo() {
    try {
        // do something that may cause an exception
    }
    catch (Exception ex){
        // Catches all exceptions
    }
    catch (java.io.IOException ex){
        // Never reached since above catches all
    }
    catch (java.io.FileNotFoundException ex){
        // Never reached since above two are caught first
    }
}
```



Notice that we arranged the **catch** blocks so that the more general **Exception** is caught first. But this is bad because ALL exceptions are subclasses of **Exception**. That means, regardless of what type of exception occurs in the **try** block, the "first" **catch** block will ALWAYS match and therefore ALWAYS be evaluated. The remaining to **catch** blocks will never be evaluated. In fact, the JAVA compiler will detect this and tell you that the last two **catch** blocks are "unreachable". To fix the problem, we can simply reverse the order of the **catch** blocks.

An additional **finally** block may be optionally used after a set of **catch** blocks:

```
try {
    ...
}
catch (java.io.IOException ex){ ... }
catch (Exception ex){ ... }
finally {
    // Code to release resources
}
```

The **finally** block is used to release resources (e.g., closing files). It is always executed. That is, if no exception occurs, it is executed immediately after the **try** block, even if the **try** block has a **return** statement in it ! (i.e., it is executed just before returning). If an exception *does* occur, the **finally** block is executed immediately after the **catch** block is executed. If an exception occurs and no **catch** block matches, the **finally** block is evaluated before the method halts with the thrown exception.

Let us now look at what we can do inside our **catch** blocks. While inside the **catch** block, the following messages can be sent to the incoming **Exception** (i.e., to the parameter of a **catch** block):

- `getMessage ()` - returns a **String** describing the exception. Typically, these strings are short descriptions of the error.
- `printStackTrace ()` - displays the sequence of method calls that led up to the exception. This is what you see on the screen when the JVM catches an exception. This is very useful for debugging purposes.



So we can do many different things inside **catch** blocks. Here are some examples:

```
try {
    ...
}
catch (ExceptionType1 ex) {
    System.out.println("Hey! Something bad just happened!");
}
catch (ExceptionType2 ex) {
    System.out.println(ex.getMessage ());
}
catch (ExceptionType3 ex) {
    ex.printStackTrace ();
}
```

Consider the stack trace for this code:

```
import java.util.ArrayList;

public class MyClass {
    public static void doSomething(ArrayList<Integer> anArray){
        doAnotherThing(anArray);
    }
    public static void doAnotherThing(ArrayList<Integer> theArray){
        System.out.println(theArray.get(0));    // Error is generated
    }
    public static void main(String[] args){
        doSomething(null);
    }
}
```

When we run this code, we get the following stack trace printed to the console window:

```
java.lang.NullPointerException
  at MyClass.doAnotherThing(MyClass.java:7)
  at MyClass.doSomething(MyClass.java:4)
  at MyClass.main(MyClass.java:10)
```

Notice that the stack trace indicates:

1. the kind of **Exception** that was generated
2. the method that generated the exception and
3. the line number at which the exception occurred

10.3 Examples of Handling Exceptions

Let us now look at how we can handle (i.e., catch) a standard **Exception** in JAVA. Consider a program that reads in two integers and divides the first one by the second and then shows the answer. We will assume that we want the number of times that the second number divides evenly into the first (i.e., ignore the remainder). What problems can occur? Well, we may get invalid data or we may get a divide by zero error. Let us look at how we would have done this previously ...



```
import java.util.Scanner;

public class ExceptionTestProgram1 {
    public static void main(String[] args) {
        int    number1, number2, result;
        Scanner keyboard;

        keyboard = new Scanner(System.in);
        System.out.println("Enter the first number:");
        number1 = keyboard.nextInt();

        System.out.println("Enter the second number:");
        number2 = keyboard.nextInt();

        System.out.print(number2 + " goes into " + number1);
        System.out.print(" this many times: ");

        result = number1 / number2;
        System.out.println(result);
    }
}
```

Here is the output if **143** and **24** are entered:

```
Enter the first number:
143
Enter the second number:
24
24 goes into 143 this many times: 5
```

What if we now enter **143** and **ABC** ?

```
Enter the first number:
143
Enter the second number:
ABC
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:840)
    at java.util.Scanner.next(Scanner.java:1461)
    at java.util.Scanner.nextInt(Scanner.java:2091)
    at java.util.Scanner.nextInt(Scanner.java:2050)
    at ExceptionTestProgram1.main(ExceptionTestProgram1.java:13)
```

This is not a pleasant way for your program to end. By default, when exceptions occur, they actually print out the stack trace (i.e., the sequence of method calls that led to the exception). That is what we are seeing here. It is ugly, but good for debugging purposes.

Notice what happened. The first line of the stack trace indicates that an **InputMismatchException** has occurred.

The second line tells us that the error occurred at line **840** of the **Scanner.java** code from a method called **throwFor()**. This was not code that we wrote ... it is pre-existing code from JAVA's **Scanner** class. The error, however, is not in line 840 of the **Scanner** class code. That is just where the error *surfaced*.

By looking further down the stack trace, we can gain insight as to why our code caused the **Exception** to occur. We just need to look down the stack trace until we find a method that we wrote. Notice that most of the successive method calls were in the **Scanner** class. However, right at the bottom we notice that the **main** method was called.

As it turns out, JAVA is telling us that the error occurred as a result of line **13** in our **ExceptionTestProgram1**. That is the code that tries to obtain the next integer from the Scanner. When it attempts to do this, we get an "Input Mismatch" because we entered ABC when we ran the program ... and ABC cannot be converted to an integer.

So now that we know WHY the error occurred, how can we gracefully handle the error? We certainly do not want to see the stack trace message !!!

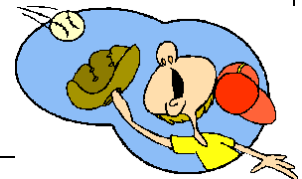
In order to handle the entering of bad data (e.g., ABC instead of an integer) we would need to do one of two things:

1. either modify the code in the **Scanner** class to detect and gracefully handle the error, or
2. catch the **InputMismatchException** within our code and gracefully handle the error.

Since it is not usually possible nor recommended to copy and start modifying the available JAVA class libraries, our best choice would be to catch and handle the error from our own code. We will have to "look for" (i.e., catch) the **InputMismatchException** by placing **try/catch** blocks in the code appropriately as follows:

```
try {
    System.out.println("Enter the first number:");
    number1 = keyboard.nextInt();

    System.out.println("Enter the second number:");
    number2 = keyboard.nextInt();
}
catch (java.util.InputMismatchException e) {
    System.out.println("Those were not proper integers! I quit!");
    System.exit(-1);
}
System.out.print(number2 + " goes into " + number1);
...
```



Notice in the **catch** block that we display an error message when the error occurs and then we do: `System.exit(-1);`. This is a quick way to halt the program completely.

The value of **-1** is somewhat arbitrary but when a program stops we need to supply some kind of integer value. Usually the value is a special code that indicates what happened. Often, programmers will use **-1** to indicate that an error occurred.

Once we incorporate the **try** block, JAVA indicates to us the following compile errors:

```
variable number2 might not have been initialized
variable number1 might not have been initialized
```

It is referring to this line:

```
System.out.print(number2 + " goes into " + number1);
```

Here we are using the **number1** and **number2** variables. However, because the **try** block may generate an error, JAVA is telling us that there is a chance that we will never assign values to these variables (i.e., they might not be initialized) and so we might obtain wrong data. JAVA does not like variables that have no values ... so it is forcing us to assign a value to these two variables. It is perhaps the most annoying type of compile error in JAVA, but nevertheless we must deal with it. The simplest way is to just assign a value of **0** to each of these variables when we declare them. Here is the updated version:

```
import java.util.Scanner;

public class ExceptionTestProgram2 {
    public static void main(String[] args) {
        int    number1 = 0, number2 = 0, result;
        Scanner keyboard;

        keyboard = new Scanner(System.in);
        try {
            System.out.println("Enter the first number:");
            number1 = keyboard.nextInt();

            System.out.println("Enter the second number:");
            number2 = keyboard.nextInt();
        }
        catch (java.util.InputMismatchException e) {
            System.out.println("Those were not proper integers! I quit!");
            System.exit(-1);
        }
        System.out.print(number2 + " goes into " + number1);
        System.out.print(" this many times: ");

        result = number1 / number2;
        System.out.println(result);
    }
}
```

If we test it again with **143** and **24** as before, it still works the same. However, now when tested with **143** and **ABC**, here is the output:

```
Enter the first number:
143
Enter the second number:
ABC
Those were not proper integers! I quit!
```

What if we enter **ABC** as the first number ?

```
Enter the first number:
ABC
Those were not proper integers! I quit!
```

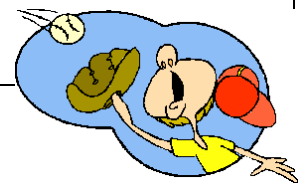
Woops! It appears that our error message is not grammatically correct anymore. Perhaps we should change it to "Invalid integer entered!" ... this should be clear enough.

Now let us test the code with values **12** and **0**:

```
Enter the first number:
12
Enter the second number:
0
0 goes into 12 this many times: Exception in thread "main"
java.lang.ArithmeticException: / by zero
    at ExceptionTestProgram2.main(ExceptionTestProgram2.java:23)
```

JAVA has detected that we tried to divide a number by zero ... a big "no no" in the world of mathematics. We can handle the **ArithmeticException** by adding additional **try/catch** blocks around line **23** of our code:

```
try {
    result = number1 / number2;
}
catch (ArithmeticException e) {
    System.out.println("Second number is 0, cannot do division!");
    System.exit(-1);
}
System.out.println(result);
```



We can merge the two **try** blocks into one if we want to as follows...


```
import java.util.Scanner;

public class ExceptionTestProgram3 {
    public static void main(String[] args) {
        int    number1 = 0, number2 = 0, result = 0;
        Scanner keyboard;

        keyboard = new Scanner(System.in);
        try {
            System.out.println("Enter the first number:");
            number1 = keyboard.nextInt();

            System.out.println("Enter the second number:");
            number2 = keyboard.nextInt();

            result = number1 / number2;
        }
        catch (java.util.InputMismatchException e) {
            System.out.println("Invalid integer entered!");
            System.exit(-1);
        }
        catch (ArithmeticException e) {
            System.out.println("Second number is 0, cannot do division!");
            System.exit(-1);
        }
        System.out.print(number2 + " goes into " + number1);
        System.out.println(" this many times: " + result);
    }
}
```

Now when we enter **12** and **0** as input, we get the appropriate message:

```
Second number is 0, cannot do division!
```

How can we adjust our code to repeatedly prompt for integers until valid ones were entered ? We would need a **while** loop since we do not know how many times to keep asking. Here is how we could do this to get a single number ...

```
int    number1 = 0;
boolean gotANumber = false;

while (!gotANumber) {
    try {
        System.out.println("Enter the first number");
        number1 = new Scanner(System.in).nextInt();
        gotANumber = true;
    }
    catch (java.util.InputMismatchException e) {
        System.out.println("Invalid integer. Please re-enter");
    }
}
```

This code would repeatedly ask for a number until it was a valid integer. However, there is a slight problem with the **Scanner** class. When the error is generated in the **Scanner** class code due to the invalid integer being entered, the **Scanner** object is messed up and is no longer ready read integers using **nextInt()**. The easiest way to fix this is to re-assign a new **Scanner** object to the **keyboard** variable when the error occurs. Here is the completed code:

```
import java.util.Scanner;

public class ExceptionTestProgram4 {
    public static void main(String[] args) {
        int    number1 = 0, number2 = 0, result = 0;
        boolean gotANumber = false;
        Scanner keyboard;

        keyboard = new Scanner(System.in);
        while(!gotANumber) {
            try {
                System.out.println("Enter the first number");
                number1 = keyboard.nextInt();
                gotANumber = true;
            }
            catch (java.util.InputMismatchException e) {
                System.out.println("Invalid integer. Please re-enter");
                keyboard = new Scanner(System.in);
            }
        }
        gotANumber = false;
        while(!gotANumber) {
            try {
                System.out.println("Enter the second number");
                number2 = keyboard.nextInt();
                gotANumber = true;
            }
            catch (java.util.InputMismatchException e) {
                System.out.println("Invalid integer. Please re-enter");
                keyboard = new Scanner(System.in);
            }
        }
        try {
            result = number1 / number2;
            System.out.print(number2 + " goes into " + number1);
            System.out.println(" this many times: " + result);
        }
        catch (ArithmeticException e) {
            System.out.println("Second number is 0, cannot do division!");
        }
    }
}
```

Here are the test results:

```
Enter the first number
what
Invalid integer. Please re-enter
Enter the first number
help me
Invalid integer. Please re-enter
Enter the first number
ok, ok, here goes
Invalid integer. Please re-enter
Enter the first number
143
Enter the second number
did you say number 2 ?
Invalid integer. Please re-enter
Enter the second number
40
40 goes into 143 this many times: 3
```

10.4 Creating and Throwing Your Own Exceptions

You may throw an exception in your code at any time if you want to inform everyone that an error occurred in **your** code. Thus, you do not need to *handle* the error in your code, you can simply *delegate* (i.e., transfer) the responsibility to whoever calls your method.

Exceptions are thrown with the **throw** statement. Basically, when we want to generate an exception, we create a new **Exception** object by calling one of its constructors, and then **throw** it as follows:

```
throw new java.io.FileNotFoundException();
throw new NullPointerException();
throw new Exception();
```

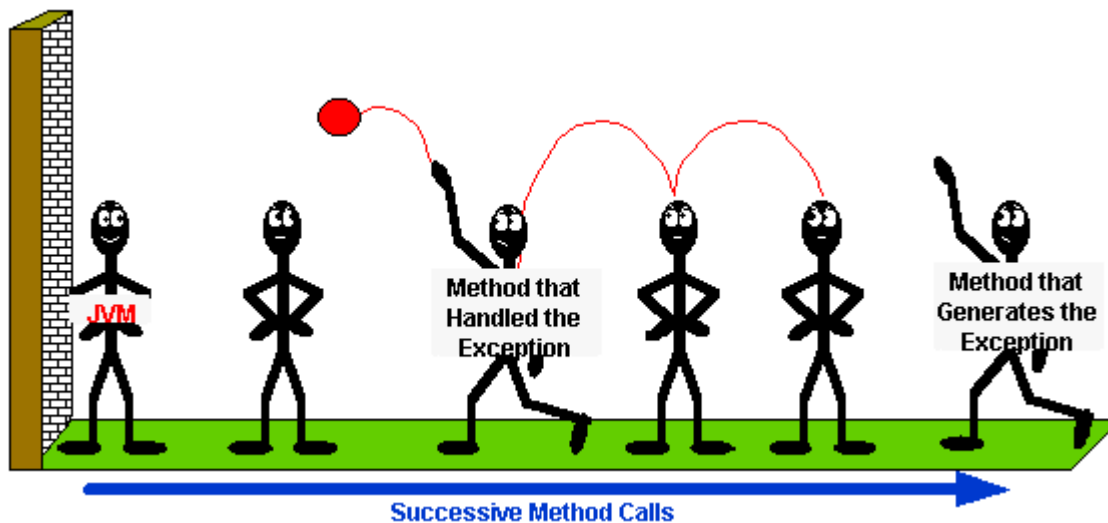
Methods that throw these exceptions, must declare that they do so in their method declarations, using the **throws** clause (as we have seen before):

```
public void yourMethod() throws anException {
    ...
}
```



You may even catch an exception, partially handle it and then throw it **again**:

```
public void yourMethod() throws Exception {
    try {
        ...
    }
    catch (Exception ex){
        ... // partially handle the exception here
        throw ex; // then throw it again
    }
}
```



Catching and then throwing an exception again is useful, for example, if we want to:

- just keep an internal "log" of errors that were generated,
- attach additional information to the exception message, or
- delay the passing on of the exception to the calling method

It is also possible to create "your own" types of exceptions. This would allow you to catch specific types of problems in your code that JAVA would normally ignore. To make your own exceptions, you simply need to create a subclass of an existing exception. If you are unsure where to put it in the hierarchy, you should use **Exception** as the superclass.

Here are the steps to making your own **Exception**:

1. choose a *meaningful* class/exception name (e.g., **WrongPasswordException**)
2. specify the superclass under which this exception will reside (e.g., **Exception**)
3. optionally provide a constructor (for simplicity, this constructor may just call the **super** constructor, passing in a string indicating the reason for the error).

Here is an example of a newly defined exception called **MyExceptionName**. We define it just as we would any other class and then save it to a file called **MyExceptionName.java**. It must also be compiled before it can be used in your program ...

```
public class MyExceptionName extends Exception {
    public MyExceptionName() {
        super("Some string explaining the exception");
    }
}
```

Consider an example of how we could force the user to type in their name (i.e., not leave it blank). We could do the following:

```
import java.util.Scanner;

public class MyExceptionTestProgram {
    public static void main(String[] args) {
        String    name = "";
        boolean    gotValidName = false;
        Scanner    keyboard = new Scanner(System.in);

        while (!gotValidName) {
            System.out.println("Enter your name");
            name = keyboard.nextLine();
            if (name.length() > 0)
                gotValidName = true;
            else
                System.out.println("Error: Name must not be blank");
        }
        System.out.println("Hello " + name);
    }
}
```

Here would be the output of such a program:

```
Enter your name
Error: Name must not be blank
Enter your name
Mark
Hello Mark
```

Notice how the “error” is detected ... we simply check the data for an empty string and use **if/else** statements to determine whether or not the error has occurred and then display an appropriate message.

In some programs, however, we may not want to print a message to the screen. For example, we may want to bring up a dialog box. In fact, we may not know exactly what to do, as it depends on our user interface as well as the context within our application. In such cases (i.e., when we are not sure what to do), we could simply generate an exception and let the method that called our code decide what to do.

Let us generate a **MissingNameException** when the user does not enter a name. We can do this by starting with our own exception definition as follows:

```
public class MissingNameException extends Exception {
    public MissingNameException() {
        super("Name is blank");
    }
}
```

We need to save and compile that code in its own file. Now, how do we generate the exception? We simply call `throw new MissingNameException()` at the right spot in the code:

```
import java.util.Scanner;

public class MyExceptionTestProgram2 {
    public static void main(String[] args) throws MissingNameException {
        String name = "";
        boolean gotValidName = false;
        Scanner keyboard = new Scanner(System.in);

        while (!gotValidName) {
            System.out.println("Enter your name");
            name = keyboard.nextLine();
            if (name.length() <= 0)
                throw new MissingNameException();
            gotValidName = true;
        }
        System.out.println("Hello " + name);
    }
}
```

Notice that we must declare in our method that we now “throw” the exception. If we run the code as before, we can see this new exception being generated:

```
Enter your name
```

```
Exception in thread "main" MissingNameException: Name is blank
at MyExceptionTestProgram2.main(MyExceptionTestProgram2.java:12)
```

Congratulations to us ... we have successfully created and generated our own exception. How though can we handle the exception? So that we may use the same example, let us adjust the code a little by creating a method that will get the user input for us as follows ...

```
public String getName() throws MissingNameException {
    String name = new Scanner(System.in).nextLine();
    if (name.length() <= 0)
        throw new MissingNameException();
    return name;
}
```

The above method gets the name from the user and returns it ... unless the name is blank ... in which case it generates the **MissingNameException**.

Now we should catch the error from our **main** program as follows:

```
import java.util.Scanner;

public class MyExceptionTestProgram3 {

    // Method to get the name from the user
    public static String getName() throws MissingNameException {
        String name = new Scanner(System.in).nextLine();
        if (name.length() <= 0)
            throw new MissingNameException();
        return name;
    }

    // Main method to test out the MissingNameException
    public static void main(String[] args) {
        String name = "";
        boolean gotValidName = false;

        while (!gotValidName) {
            System.out.println("Enter your name");
            try {
                name = getName();
                gotValidName = true;
            }
            catch (MissingNameException ex) {
                System.out.println("Error: Name must not be blank");
            }
        }
        System.out.println("Hello " + name);
    }
}
```



The resulting output is the same as before (i.e., same as `MyExceptionTestProgram`).

As another example, let us take another look at the **BankAccount** object again ... more specifically ... consider this **withdraw()** method:

```
public boolean withdraw(float anAmount) {
    if (anAmount <= this.balance) {
        this.balance -= anAmount;
        return true;
    }
    return false;
}
```



When the user tries to withdraw more money than is actually in the account ... nothing happens. Since the method returns a **boolean**, we can always check for this error where we call the method:

```

public static void main(String[] args) {
    BankAccount b = new BankAccount("Bob");
    b.deposit(100);
    b.deposit(500.00f);

    if (!b.withdraw(25.00f))
        System.out.println("Error withdrawing money from account");
    if (!b.withdraw(189.45f))
        System.out.println("Error withdrawing money from account");
    b.deposit(100.00f);
    if (!b.withdraw(1000000))
        System.out.println("Error withdrawing money from account");
}

```

This form of error checking works fine, but it clearly clutters up the code! Let us see how we can make use of an Exception. We will create a **WithdrawalException** object. Where would it go in the **Exception** hierarchy? Probably right under the **Exception** class again, since there are no existing bank-related exception classes in JAVA. Here is the exception:

```

public class WithdrawalException extends Exception {
    public WithdrawalException() {
        super("Error making withdrawal");
    }
}

```

Now how do we *throw* the exception from within the **withdraw()** method? Here is how we do it ...

```

public void withdraw(float anAmount) throws WithdrawalException {
    if (anAmount <= this.balance)
        this.balance -= anAmount;
    else
        throw new WithdrawalException();
}

```

Note that we must also instruct the compiler that this method may throw a **WithdrawalException** by writing this as part of the method declaration. The addition of this simple statement will force all methods that call the **withdraw()** method to deal with the exception.

Also notice that we no longer need the **boolean** return type for the **withdraw()** method since its purpose was solely for error checking. Now that we have the exception being generated, this becomes our new form of error checking.

Now how do we change the code that calls the **withdraw()** method? We just need to enclose our withdrawal code in a **try** block:


```
public static void main(String[] args) {
    BankAccount b = new BankAccount("Bob");
    try {
        b.deposit(100);
        b.deposit(500.00f);
        b.withdraw(25.00f);
        b.withdraw(189.45f);
        b.deposit(100.00f);
        b.withdraw(1000000);
    } catch (WithdrawalException ex) {
        System.out.println("Error withdrawing money");
    }
}
```

Notice how much simpler and cleaner the calling code becomes. Be aware however, that if one error occurs early within the **try** block, none of the remaining code in the **try** block gets evaluated!!! So an error in the first withdrawal attempt would prevent the two other withdrawals and deposit being made on the account from happening. If we did not want this behavior, we would need to make a separate **try/catch** block for each of the 3 **withdraw()** method calls.

We can make our code even simpler by ignoring the error. To do this we would have to indicate in the **main()** method that the **WithdrawalException** may occur as follows ...

```
public static void main(String[] args) throws WithdrawalException {
    BankAccount b = new BankAccount("Bob");
    b.deposit(100);
    b.deposit(500.00f);
    b.withdraw(25.00f);
    b.withdraw(189.45f);
    b.deposit(100.00f);
    b.withdraw(1000000);
}
```

If we do this, however, then the program will stop and quit when the first **WithdrawalException** occurs.

We can actually add more information to our exceptions. For example, there may be many reasons why we cannot withdraw from a **BankAccount**. The bank account ...

- may not have enough money in it,
- may not allow withdrawals (e.g., some kinds of **SavingsAccounts**), or
- may not have sufficient funds to cover transaction fees associated with the account

We do not need to make different types of exceptions, but can instead supply more information when the **WithdrawException** is generated. The easiest way to do this is to modify the constructor in our **WithdrawalException** class that takes a **String** parameter to describe the error:

```
public class WithdrawalException extends Exception {
    public WithdrawalException(String description) {
        super(description);
    }
}
```

We can then use this new constructor instead by supplying different explanations as to why the error occurred. For example, the **SuperSavings** account may have the following **withdraw()** method:

```
public void withdraw(float anAmount) throws WithdrawalException {
    throw new WithdrawalException("Withdrawals not allowed from this account");
}
```

whereas the **PowerSavings** account may have this method ...

```
public void withdraw(float anAmount) throws WithdrawalException {
    if (anAmount > this.balance)
        throw new WithdrawalException("Insufficient funds in account to
            withdraw specified amount");
    if (anAmount + WITHDRAW_FEE > this.balance) {
        throw new WithdrawalException("Not enough money to cover
            transaction fee");
    }
    this.balance -= anAmount + WITHDRAW_FEE;
}
```

So, as can easily be seen, we can provide additional explanatory information for the user when an exception does occur. Furthermore, we can do this with a single exception class (i.e., we do not need to make a subclass of **WithdrawalException** for each specific situation).

We can extract this “additional explanation” from the exception by sending the **getMessage()** message to the exception within our **catch** blocks:

```
public static void main(String[] args) {
    PowerSavings p = new PowerSavings("Bob");
    SuperSavings s = new SuperSavings("Betty");
    try {
        p.deposit(100);
        s.deposit(500.00f);
        p.withdraw(25.00f);
        p.withdraw(189.45f);
        s.deposit(100.00f);
        s.withdraw(1000000);
    } catch (WithdrawalException ex) {
        System.out.println(ex.getMessage());
    }
}
```



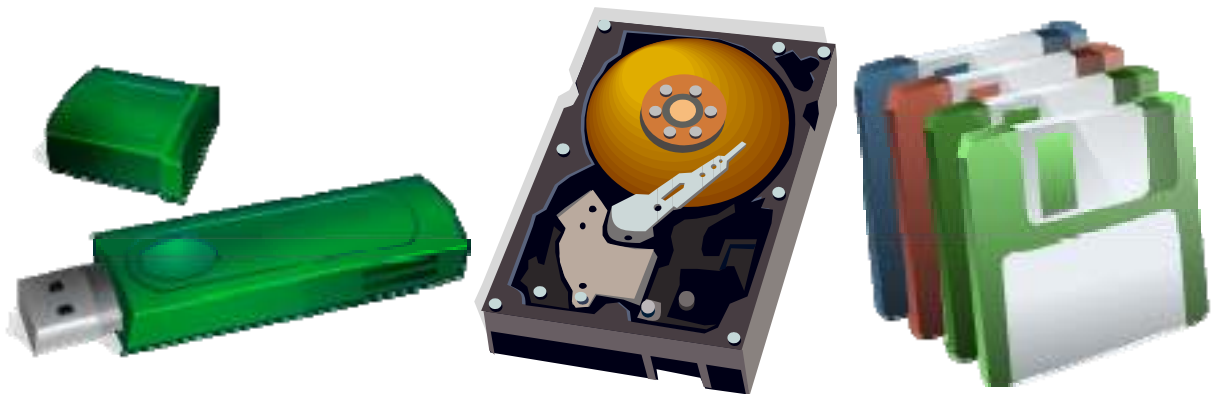
In this example, the **catch** block catches any errors for both bank accounts.

Chapter 11

Saving and Loading Information

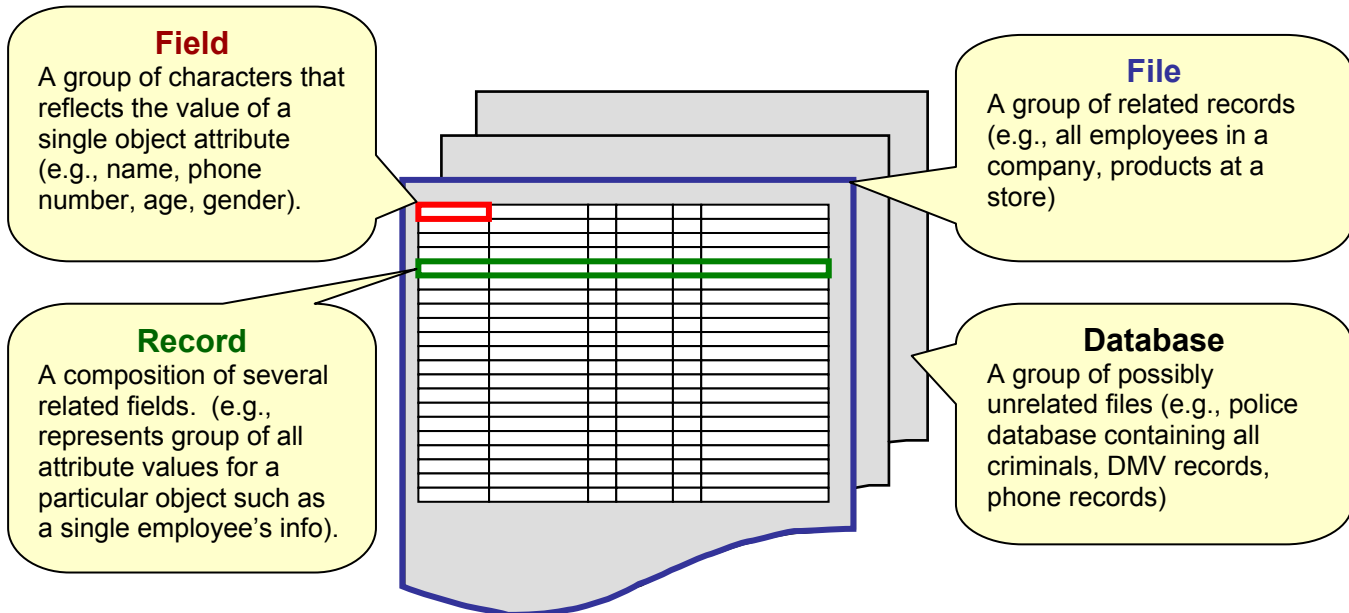
What is in This Chapter ?

In computer science, all data eventually gets stored onto storage devices such as hard drives, diskettes, USB flash drives, CDs, DVDs, etc... This set of notes explains how to save information from your program to a file that sits on one of these backup devices. It also discusses how to load that information back into your program. The saving/loading of data from files can be done using different formats. We discuss here the notion of **text** vs. **binary** formats. Note as well that the techniques presented here also apply to sending and receiving information from **Streams** (e.g., networks). We will look at the way in which **Stream** objects are used to do data I/O in JAVA. We will also look at how to use **ObjectStreams** to read/write entire objects easily and finally investigate the **File** class which is useful for querying files and folders on your computer.



11.1 Introduction to Files and Streams

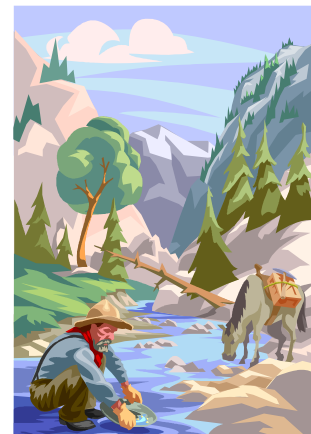
File processing is very important since eventually, all data must be stored externally from the machine so that it will not be erased when the power is turned off. Here are some of the terms related to file processing:



In JAVA, we can store information from our various objects by extracting their attributes and saving these to the file. To use a file, it must be first **opened**. When done with a file, it **MUST** be **closed**. We use the terms **read** to denote getting information *from* a file and **write** to denote saving information *to* a file. The contents of a file is ultimately reduced to a set of numbers from 0 to 255 called **bytes**.

In JAVA, files are represented as **Stream** objects. The idea is that data “streams” (or flows) to/from the file ... similar to the idea of streaming video that you may have seen online. Streams are **objects** that allow us to send or receive information in the form of bytes. The information that is put into a stream, comes out in the same order.

It is similar to those scrolling signs where the letters scroll from right to left, spelling out a sentence:



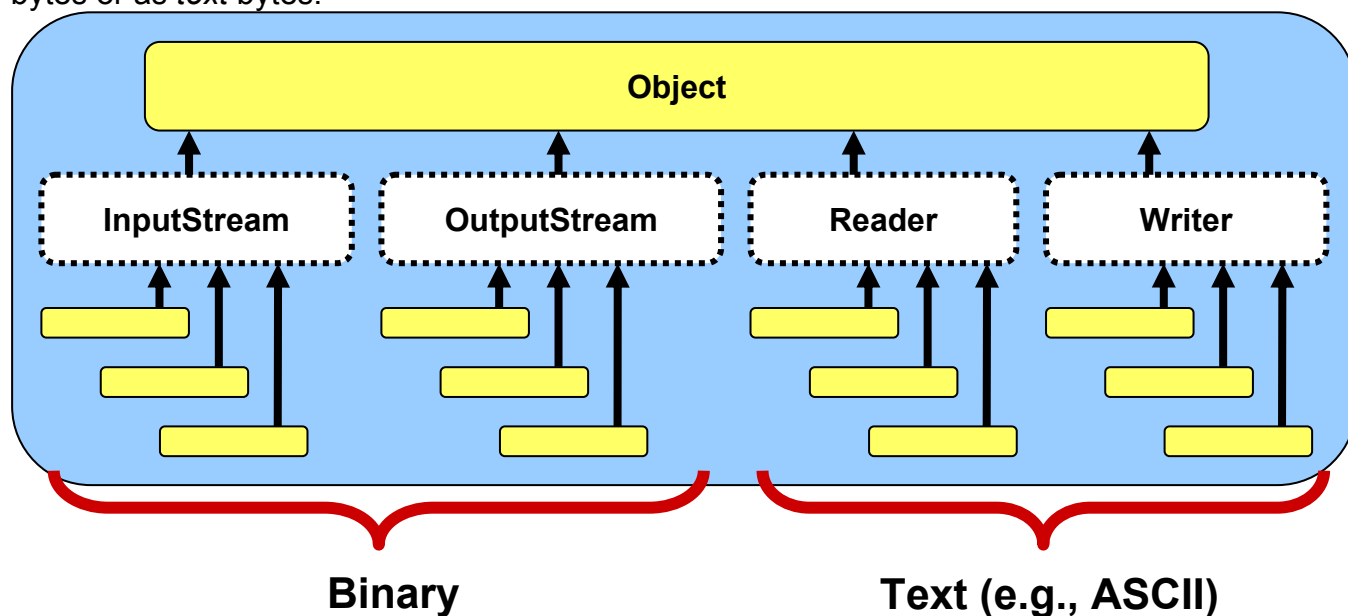
Streams are actually very general in that they provide a way to send or receive information to and from:

- files
- networks
- different programs
- any I/O devices (e.g., console and keyboard)

When we first start executing a JAVA program, 3 streams are automatically created:

- `System.in` // for inputting data from the keyboard
- `System.out` // for outputting data to the screen
- `System.err` // for outputting error messages to the screen

In fact, there are many stream-related classes in JAVA. We will look at a few and how they are used to do file I/O. The various **Streams** differ in the way that data is “entered into” and “extracted from” the stream. As with **Exceptions**, **Streams** are organized into different hierarchies. JAVA contains four main stream-related hierarchies for transferring data as *binary* bytes or as *text* bytes:



It is interesting to note that there is no common **Stream** class from which these main classes inherit. Instead, these 4 **abstract** classes are the root of more specific subclass hierarchies. A rather large number of classes are provided by JAVA to construct streams with the desired properties. We will examine just a few of the common ones here.

Typically I/O (i.e., input/output) is a bottleneck in many applications. That is, it is very time consuming to do I/O operations when compared to internal operations. For this reason, **buffers** are used. Buffered output allows data to be collected for output before it is actually sent to the output device. Only when the buffer gets full does the actual data get sent. This reduces the amount of actual output operations, but each output operation would usually send more data. (Note: The **flush()** command can be sent to buffered streams in order to empty the buffer and cause the data to be sent "immediately" to the output device. Input data can also be buffered.)

By the way, what is **System.in** and **System.out** exactly ? We can determine their respective classes with the following code:

```
System.out.print("System.in is an instance of ");
System.out.println(System.in.getClass());
System.out.print("System.out is an instance of ");
System.out.println(System.out.getClass());
```

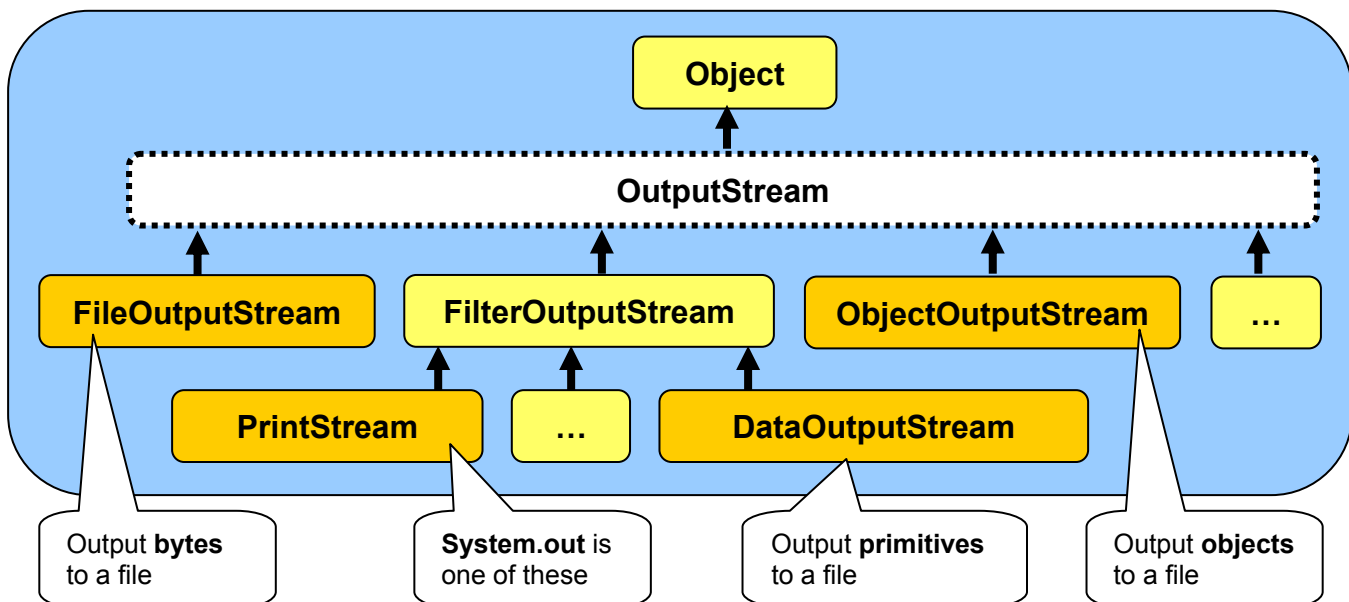
This code produces the following output:

```
System.in is an instance of class java.io.BufferedInputStream
System.out is an instance of class java.io.PrintStream
```

So we have been using these streams for displaying information and getting information from the user through the keyboard. We will now look at how the classes are arranged in the different stream sub-hierarchies.

11.2 Reading and Writing Binary Data

First, let us examine a portion of JAVA's **OutputStream** sub-hierarchy:



The streams in this sub-hierarchy are responsible for outputting binary data. That is, data which is in the form of bytes or data types. **OutputStreams** have a **write()** method that allows us to output a single byte of data at a time.

To open a file for binary writing, we can create an instance of **FileOutputStream** using one of the following constructors:

```
new FileOutputStream(String fileName);
new FileOutputStream(String fileName, boolean append);
```

The first constructor opens a new file output stream with that name. If one exists already with that name, it is overwritten (i.e., erased). The second constructor allows you to determine whether you want an existing file to be overwritten or appended to. If the file does not exist, a new one with the given name is created. If the file already exists prior to opening then the following rules apply:

- if `append = false` the existing file's contents is discarded and the file will be overwritten.
- if `append = true` the new data to be written to the file is appended to the end of the file.

We can output simple bytes to a **FileOutputStream** by using the **write()** method, which takes a single byte (i.e., a number from 0 to 255) as follows:

```
FileOutputStream out;

out = new FileOutputStream("myFile.dat");
out.write('H');
out.write(69);
out.write(76);
out.write('L');
out.write('O');
out.write('!');
out.close();
```

This code outputs the characters **HELLO!** to a file called **"myFile.dat"**. The file will be created (if not existing already) in the **current** directory/folder (i.e., the directory/folder that your JAVA program was run from). Alternatively, you can specify where to create the file by specifying the whole path name instead of just the file name as follows:

```
FileOutputStream out;
out = new FileOutputStream("F:\\My Documents\\myFile.dat");
```

Notice the use of "two" backslash characters within the String constant (because the backslash character is a special character which requires it to be preceded by a backslash ... just like `\n` is used to create a new line).

Using this strategy, we can output either characters or positive **integers** in the range from 0 to 255. Notice in the code that we closed the file stream when done. This is important to ensure that the operating system (e.g., Windows 7) releases the file handles correctly.

When working with files in this way, two exceptions may occur:

- opening a file for reading or writing may generate a [java.io.FileNotFoundException](#)
- reading or writing to/from a file may generate a [java.io.IOException](#)

You should handle these exceptions with appropriate **try/catch** blocks:

```
import java.io.*;

public class FileOutputStreamTestProgram {
    public static void main(String[] args) {
        try {
            FileOutputStream out;
            out = new FileOutputStream("myFile.dat");
            out.write('H'); out.write(69);
            out.write(76); out.write('L');
            out.write('O'); out.write('!');
            out.close();

        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for writing");
        } catch (IOException e) {
            System.out.println("Error: Cannot write to file");
        }
    }
}
```

Since all streams are a part of the **java.io** package we need to import them.

The code above allows us to output any data as long as it is in **byte** format. This can be tedious. For example, if we have the integer 7293901 and we want to output it, we have a few choices:

- break up the integer into its 7 digits and output these digits one at a time (very tedious)
- output the 4 bytes corresponding to the integer itself (recall that an **int** is stored as 4 bytes)

Either way, these are not fun. Fortunately, JAVA provides a **DataOutputStream** class which allows us to output whole primitives (e.g., **ints**, **floats**, **doubles**) as well as whole **Strings** to a file! Here is an example of how to use it to output information from a **BankAccount** object ...


```

import java.io.*;

public class DataOutputStreamTestProgram {
    public static void main(String[] args) {
        try {
            BankAccount      aBankAccount;
            DataOutputStream  out;

            aBankAccount = new BankAccount("Rob Banks");
            aBankAccount.deposit(100);

            out = new DataOutputStream(new FileOutputStream("myAcc.dat"));
            out.writeUTF(aBankAccount.getOwner());
            out.writeInt(aBankAccount.getAccountNumber());
            out.writeFloat(aBankAccount.getBalance());
            out.close();

        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for writing");
        } catch (IOException e) {
            System.out.println("Error: Cannot write to file");
        }
    }
}

```

The **DataOutputStream** acts as a “wrapper” class around the **FileOutputStream**. It takes care of breaking our primitive data types and Strings into separate bytes to be sent to the **FileOutputStream**.

There are methods to write each of the primitives as well as **Strings**:

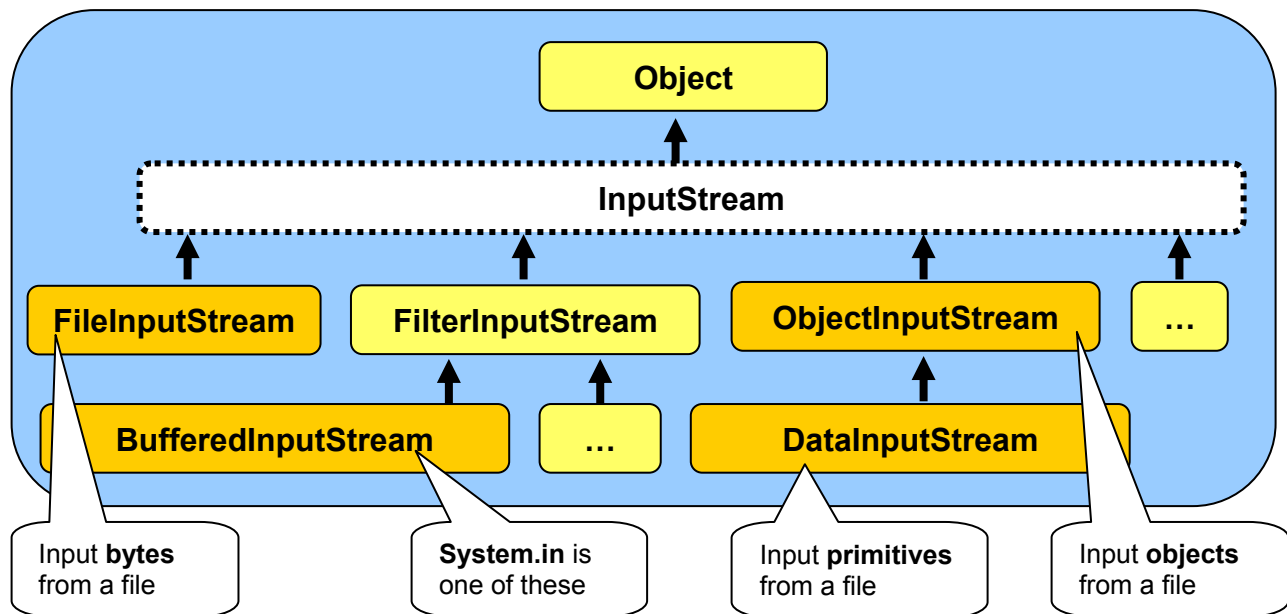
<code>writeUTF(String aString)</code>	<code>writeShort(short aShort)</code>
<code>writeInt(int anInt)</code>	<code>writeBoolean(boolean aBool)</code>
<code>writeFloat(float aFloat)</code>	<code>writeByte(int aByte)</code>
<code>writeLong(long aLong)</code>	<code>writeChar(char aChar)</code>
<code>writeDouble(double aDouble)</code>	

The output from a **DataOutputStream** is not very nice to look at (i.e., it is in binary format). The **myAcc.dat** file would display as follows if we tried to view it with Windows Notepad:

```
Rob Banks + BÈ
```

This is the binary representation of the data, which usually takes up less space than text files. The disadvantage of course, is that we cannot make sense of the data if we try to read it with our eyes as text. However, rest assured that the data is saved properly.

Let us now examine how we could read that information back in from the file with a different program. To start, we need to take a look at the **InputStream** sub-hierarchy as follows ...



Notice that it is quite similar to the **OutputStream** hierarchy. In fact, its usage is also very similar. We can read back in the byte data from our file by using **FileInputStream** now as follows:

```

import java.io.*;

public class FileInputStreamTestProgram {
    public static void main(String[] args) {
        try {
            FileInputStream in = new FileInputStream("myFile.dat");
            while(in.available() > 0)
                System.out.print(in.read() + " ");
            in.close();
        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for reading");
        } catch (IOException e) {
            System.out.println("Error: Cannot read from file");
        }
    }
}

```

Notice that we now use **read()** to read in a single byte from the file. Notice as well that we can use the **available()** method which returns the number of bytes available to be read in from the file (i.e., the file size minus the number of bytes already read in).

The code reads the data back in from our file (i.e., the characters **HELLO!**) and outputs their ASCII (i.e., byte) values to the console:

```
72 69 76 76 79 33
```

Try changing `in.read()` to `(char)in.read()` (i.e., type-cast the byte to a char) ... what happens ?

That was fairly simple ... but what about getting back those primitives ? You guessed it! We will use **DataInputStream**:

```
import java.io.*;

public class DataInputStreamTestProgram {
    public static void main(String[] args) {
        try {
            BankAccount    aBankAccount;
            DataInputStream in;

            in = new DataInputStream(new FileInputStream("myAccount.dat"));

            String name = in.readUTF();
            int    acc = in.readInt();
            float  bal = in.readFloat();

            aBankAccount = new BankAccount(name, bal, acc);
            System.out.println(aBankAccount);
            in.close();

        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for reading");
        } catch (IOException e) {
            System.out.println("Error: Cannot read from file");
        }
    }
}
```

Notice that we re-create a new **BankAccount** object and "fill-it-in" with the incoming file data. Note, that in order for the above code to compile, we would need to write a public constructor for the **BankAccount** class that takes an owner name, balance and account number (i.e., previously, in our **BankAccount** class, we had no way of specifying the **accountNumber** since it was set automatically) ...

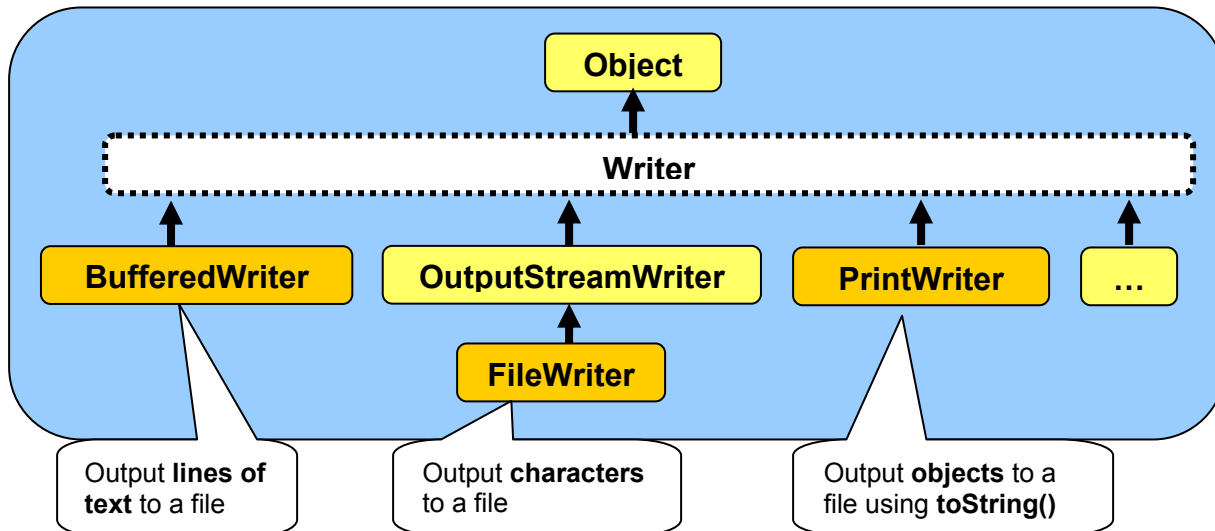
```
BankAccount(String initName, float initBal, int num) {
    ownerName = initName;
    accountNumber = num;
    balance = initBal;
}
```

As with the output streams, there are methods to read in the other primitives:

String	<code>readUTF()</code>	short	<code>readShort()</code>
int	<code>readInt()</code>	boolean	<code>readBoolean()</code>
float	<code>readFloat()</code>	int	<code>readByte()</code>
long	<code>readLong()</code>	char	<code>readChar()</code>
double	<code>readDouble()</code>		

11.3 Reading and Writing Text Data

Here is the **Writer** class sub-hierarchy which is used for writing **text** data to a stream:



Notice that there are 3 main classes we will use for writing **characters**, **lines of characters** and general **objects** to a text file. When objects are written to the text file, the **toString()** method for the object is called and the resulting **String** is saved to the file.

We can output (in text format) to a file using simply the **print()** or **println()** methods with the **PrintWriter** class as follows ...

```

import java.io.*;

public class PrintWriterTestProgram {
    public static void main(String[] args) {
        try {
            BankAccount aBankAccount;
            PrintWriter out;

            aBankAccount = new BankAccount("Rob Banks");
            aBankAccount.deposit(100);

            out = new PrintWriter(new FileWriter("myAccount2.dat"));

            out.println(aBankAccount.getOwner());
            out.println(aBankAccount.getAccountNumber());
            out.println(aBankAccount.getBalance());
            out.close();

        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for writing");
        } catch (IOException e) {
            System.out.println("Error: Cannot write to file");
        }
    }
}

```

Wow! Outputting text to a file is as easy as outputting it to the console window ! But what does it look like ? If we opened the file with Windows Notepad, we would notice that the result is a “pleasant looking” text format:

```
Rob Banks
100000
100.0
```

In fact, we can actually write any object using the **println()** method. JAVA will use that object's **toString()** method. So if we replaced this code:

```
out.println(aBankAccount.getOwner());
out.println(aBankAccount.getAccountNumber());
out.println(aBankAccount.getBalance());
```

with this code:

```
out.println(aBankAccount);
```

we would end up with the following saved to the file:

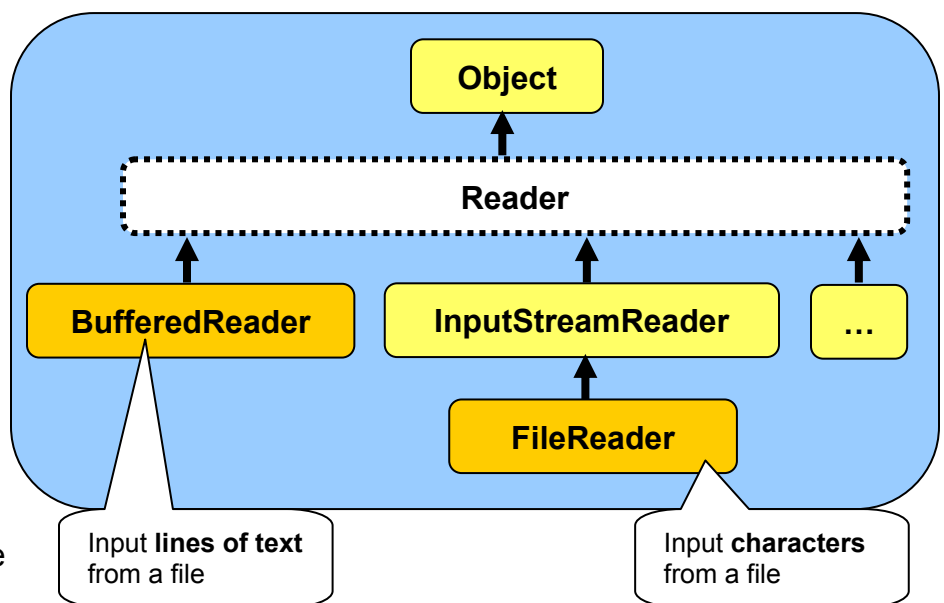
```
Account #100000 with $100.0
```

So it actually does behave just like the **System.out** console. We would need to be careful though, because you will notice that the **BankAccount's toString()** method in the example above did not display the owner's name. So the file does not record that owner's name and therefore we could never read that name back in again ... it would be lost forever. Notice as well how the **PrintWriter** wraps the **FileWriter** class just as the **DataOutputStream** wrapped the **FileOutputStream**.

It is also easy to read back in the information that was saved to a text file. Here is the hierarchy of classes for reading text files →

Notice that we can only read in *characters* and *lines of characters* from the text file, but NOT general objects. We will see later how we can re-build read-in objects.

Most of the time, we will make use of the **BufferedReader** class by using the **readLine()** method as follows:



```

import java.io.*;

public class BufferedReaderTestProgram {
    public static void main(String args[]) {
        try {
            BankAccount    aBankAccount;
            BufferedReader  in;

            in = new BufferedReader(new FileReader("myAccount2.dat"));
            String name = in.readLine();
            int    acc = Integer.parseInt(in.readLine());
            float  bal = Float.parseFloat(in.readLine());

            aBankAccount = new BankAccount(name, bal, acc);
            System.out.println(aBankAccount);
            in.close();

        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for reading");
        } catch (IOException e) {
            System.out.println("Error: Cannot read from file");
        }
    }
}

```

Note the use of "primitive data type" wrapper classes to read data types. We could have used the **Scanner** class here to simplify the code:

```

import java.io.*;
import java.util.*; // Needed for use of Scanner and NoSuchElementException

public class BufferedReaderTestProgram2 {
    public static void main(String[] args) {
        try {
            BankAccount    aBankAccount;

            Scanner in = new Scanner(new FileReader("myAccount2.dat"));
            String name = in.nextLine();
            int    acc = in.nextInt();
            float  bal = in.nextFloat();
            aBankAccount = new BankAccount(name, bal, acc);
            System.out.println(aBankAccount);
            in.close();

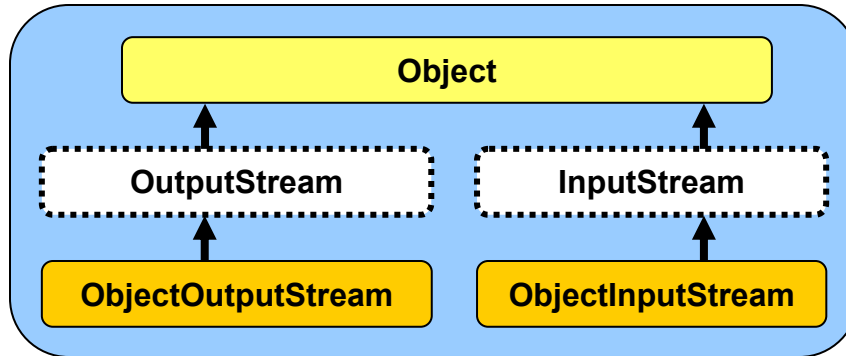
        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for reading");
        } catch (NoSuchElementException e) {
            System.out.println("Error: EOF encountered, file may be corrupt");
        } catch (IOException e) {
            System.out.println("Error: Cannot read from file");
        }
    }
}

```

Notice here that we now catch a **NoSuchElementException**. This is how the **Scanner** detects the end of the file. The main advantage of using this **Scanner** class is that we do not have to use any wrapper classes to convert the input strings to primitives.

11.4 Reading and Writing Whole Objects

So far, we have seen ways of saving and loading bytes and characters to a file. Also, we have seen how **DataOutputStream/DataInputStream** and **PrintWriter/BufferedReader** classes can make our life simpler since they deal with larger (more manageable) chunks of data such as primitives and Strings. We also looked at how we can save a whole object (i.e., a **BankAccount**) to a file by extracting its attributes and saving them individually. Now we will look at an even simpler way to save/load a whole object to/from a file using the **ObjectInputStream** and **ObjectOutputStream** classes:



These classes allow us to save or load entire JAVA objects with one method call, instead of having to break apart the object into its attributes. Here is how we do it:

```

import java.io.*;

public class ObjectOutputStreamTestProgram {
    public static void main(String args[]) {
        try {
            BankAccount          aBankAccount;
            ObjectOutputStream    out;

            aBankAccount = new BankAccount("Rob Banks");
            aBankAccount.deposit(100);

            out = new ObjectOutputStream(new FileOutputStream("myAcc.dat"));
            out.writeObject(aBankAccount);
            out.close();

        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for writing");
        } catch (IOException e) {
            System.out.println("Error: Cannot write to file");
        }
    }
}
  
```

Wow! It is VERY easy to write out an object. We simply supply the object that we want to save to the file as a parameter to the **writeObject()** method. Notice that the **ObjectOutputStream** class is a wrapper around the **FileOutputStream**. That is because ultimately, the object is reduced to a set of bytes by the **writeObject()** method, which are then saved to the file.

Serialization is the process of breaking down an object into bytes.

Thus, when an object is saved to a file, it is automatically de-constructed into bytes, these bytes are then saved to a file, and then the bytes are read back in later and the object is re-constructed again. This is all done automatically by JAVA, so we don't have to be too concerned about it.

In order to be able to save an object to a file using the **ObjectOutputStream**, the object must be **serializable** (i.e., able to be serialized...or reduced to a set of bytes). To do this, we need to inform JAVA that our object implements the **java.io.Serializable** interface as follows ...

```
public class BankAccount implements java.io.Serializable {  
    ...  
}
```

This particular interface does not actually have any methods within it that we need to implement. Instead, it merely acts as a "flag" that indicates your permission for this object to be serialized. It allows a measure of security for our objects (i.e., only **serializable** objects are able to be broken down into bytes and sent to files or over the network).

Most standard JAVA classes are **serializable** by default and so they can be saved/loaded to/from a file in this manner. When allowing our own objects to be serialized, we must make sure that all of the "pieces" of the object are also **serializable**. For example, assume that our **BankAccount** is defined as follows:

```
public class BankAccount implements java.io.Serializable {  
    Customer    owner;  
    float       balance;  
    int         accountNumber;  
    ...  
}
```

In this case, since owner is not a **String** but a **Customer** object, then we must make sure that **Customer** is also **Serializable**:

```
public class Customer implements java.io.Serializable {  
    ...  
}
```

We would need to then check whether **Customer** itself uses other objects and ensure that they too are **serializable** ... and so on. To understand this, just think of a meat grinder. If some hard marbles were placed within our meat, we cannot expect it to come out through the grinder since they cannot be reduced to a smaller form. Similarly, if we have any non-**serializable** objects in our original object, we cannot properly serialize the object.



So what does a serialized object look like anyway? Here is what the file would look like from our previous example if opened in Windows Notepad:

```
-í sr BankAccount"ËSõñúä -----I accountNumberF balanceL ownert
Ljava/lang/String;xp † BË t          Rob Banks
```

Weird ... it seems to be a mix of binary and text. As it turns out, JAVA saves all the attribute information for the object, including their types and values, as well as some other information. It does this in order to be able to re-create the object when it is read back in.

The object can be read back in by using the **readObject()** method in the **ObjectInputStream** class as follows:

```
import java.io.*;

public class ObjectInputStreamTestProgram {
    public static void main(String[] args) {
        try {
            BankAccount          aBankAccount;
            ObjectInputStream in;

            aBankAccount = new BankAccount("Rob Banks");
            aBankAccount.deposit(100);

            in = new ObjectInputStream(new FileInputStream("myAcc.dat"));
            aBankAccount = (BankAccount)in.readObject();

            System.out.println(aBankAccount);
            in.close();

        } catch (ClassNotFoundException e) {
            System.out.println("Error: Object's class does not match");
        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for writing");
        } catch (IOException e) {
            System.out.println("Error: Cannot read from file");
        }
    }
}
```

Note, that the **ObjectInputStream** wraps the **FileInputStream**. Also, notice that once read in, the object must be *type-casted* to the appropriate type (in this case **BankAccount**). Also, if there is any problem trying to re-create the object according to the type of object that we are loading, then a **ClassNotFoundException** may be generated, so we have to handle it. Finally, in order for this to work, you must also make sure that your object (i.e., **BankAccount**) has a zero-parameter constructor, otherwise an **IOException** will occur when JAVA tries to rebuild the object. Although not shown in our example above, you may also make use of the **available()** method to determine whether or not the end of the file has been reached.

Although this method is extremely easy to use, there is a potentially disastrous disadvantage. The object that is saved to the file using this strategy is actually saved in binary format which depends on the class name, the object's attribute types and names as well as the method signatures and their names. So if you change the class definition after it has been saved to the file, it may not be able to be read back in again !!! Some changes to the class do not cause problems such as adding an attribute or changing its access modifiers. So as a warning, when saving objects to a file using this strategy, you should always keep a backed-up version of all of your code so that you will be able to read these files with this backed-up code in the future.



Supplemental Information (Disguising Serialized Data)

You can actually write your own methods for serializing your objects. One reason for doing this may be to encrypt some information beforehand (such as a password). You can decide which parts of the object will be serialized and which parts will not. You can declare any object attribute as being *transient* (which means that it will not be serialized) as follows:

```
private transient String password;
```

This will tell JAVA that you do not want the password saved automatically upon serialization. That way you can write your own method to encrypt it before it is serialized.

To do this, you would need to write two methods called **writeObject(ObjectOutputStream)** and **readObject(ObjectInputStream)**. These methods will automatically be called by JAVA upon serialization and they override the default writing behavior. In fact, there are **defaultWriteObject()** and **defaultReadObject()** methods which do the default serialization behavior (i.e., the serializing before you decided to do your own). Here are examples of what you can do:

```
public void writeObject(ObjectOutputStream out) throws IOException {
    out.defaultWriteObject();
    // ... do extra stuff here to append to end of file
    out.writeObject(myField.encrypt());
}
public void readObject(ObjectInputStream in) throws IOException,
    ClassNotFoundException {
    in.defaultReadObject();
    // ... do extra stuff here to read from end of file
    myField = ((myFieldType)in.readObject()).decrypt();
}
```

11.5 Saving and Loading Example

Let us now consider a real example that shows how to save and load information from an **Autoshow** which contains **Car** objects. The **Autoshow** and **Car** classes will be defined as shown below:

```
public class Car {
    private String    make;
    private String    model;
    private String    color;
    private int       topSpeed;
    private boolean   has4Doors;

    public Car() {
        this("", "", "", 0, false);
    }

    public Car(String mak, String mod, String col, int tsp, boolean fd) {
        make = mak;
        model = mod;
        color = col;
        topSpeed = tsp;
        has4Doors = fd;
    }

    public String toString() {
        String s = color;
        if (has4Doors)
            s += " 4-door ";
        else
            s += " 2-door ";
        return (s + make + " " + model +
                " with top speed " + topSpeed + "kmph");
    }
}
```

```
import java.util.ArrayList;

public class Autoshow {
    private String    name;
    private ArrayList<Car> cars;

    public Autoshow(String n) {
        name = n;
        cars = new ArrayList<Car>();
    }

    public void addCar(Car c) {
        cars.add(c);
    }

    public void showCars() {
        for (Car c: cars)
            System.out.println(c);
    }
}
```

We will save the autoshow's information in a text file so that we can print it out or read it easily. So, we will be using the **PrintWriter** and **BufferedReader** classes. We need to decide how to format the text in the file. We will need to save the **name** of the autoshow as well as the individual **cars** in the show. For each car, the file should show the **make, model, color, topSpeed** and **has4Doors**. Perhaps we will separate the cars by a blank line to indicate that one car's data ends and another's begins as follows:

```
AutoRama 2009

Porsche
959
Red
240
false

Pontiac
Grand-Am
White
160
true

... etc ...
```

This seems like a reasonable way to save the autoshow so that the data is readable in a text program. You will notice that each **Car** object is saved the same way. Hence, it would be good to start by writing some methods that can save/load **Car** objects.

We can write the following method in the **Car** class to begin...

```
public void saveTo(PrintWriter aFile) {
    ...
}
```

Notice that the method will take a single parameter which is a **PrintWriter** object to represent the file that we are saving to. Where does this file come from? It actually does not matter. When writing this method, we should just assume that someone opened a file and handed it to us and now it is our job to write the **Car** information to the file specified through this incoming parameter. Note as well, that since we did not open the file (i.e., the **PrintWriter** was handed to us), we should also not close the file. It is the opener's responsibility to close it.

So, now how do we write the **Car** information to the file? We simply do it as if we were writing to the **System** console:

```
public void saveTo(PrintWriter aFile) {
    aFile.println(make);
    aFile.println(model);
    aFile.println(color);
    aFile.println(topSpeed);
    aFile.println(has4Doors);
}
```

That was easy. Remember though, that in order for JAVA to recognize the **PrintWriter** object, we will need to import **java.io.PrintWriter** at the top of our **Car** class. In fact, as you will see soon, we will need more classes from the **java.io** package, so it would be best to simply **import java.io.***;

The method for loading a **Car** back in from the file is also quite easy. Again, it should read from a file (i.e., a **BufferedReader** object) that is passed in as a parameter, not a file that **we** open or close. Then all we need to do is to read the information from the file. But what do we “do” with the information once it has been read in? Probably, we return it from the method so that whoever called this “load” method can decide what to do with the loaded car information. So, the method should probably return a **Car** object. Here is the method that we will write:

```
public static Car loadFrom(BufferedReader aFile) {  
    ...  
}
```

Notice that the method is **static**. This is not required, but it allows us to call the method as follows:

```
Car c = Car.loadFrom(aFile);
```

instead of doing this:

```
Car c = new Car().loadFrom(aFile);
```

That is the only difference. The **static** version is more logical. So ... now ... what goes into the method? Well, we need to at least create and return a new **Car** object, so we can start with that:

```
public static Car loadFrom(BufferedReader aFile) {  
    Car loadedCar = new Car();  
    // ...  
    return loadedCar;  
}
```

To read in the car, we should recall that we use **readLine()** to read a single line of text from a **BufferedReader** file. We need to read the lines of text in the same order that they were outputted (i.e., **make**, **model**, **color**, **topSpeed** and then **has4Doors**). Then we can set the attributes of the **Car** to the data that was read in. Here is the code:

```
public static Car loadFrom(BufferedReader aFile) throws IOException {  
    Car loadedCar = new Car();  
  
    loadedCar.make = aFile.readLine();  
    loadedCar.model = aFile.readLine();  
    loadedCar.color = aFile.readLine();  
    loadedCar.topSpeed = Integer.parseInt(aFile.readLine());  
    loadedCar.has4Doors = Boolean.parseBoolean(aFile.readLine());  
  
    return loadedCar;  
}
```

Notice that the method may throw an **IOException** (due to the fact that JAVA's **readLine()** method declares that it throws an **IOException**). We could have caught the exception here and handled it. However, since this method is just a helper method in a larger application, we are unsure what to do here if an error occurs. Therefore, by declaring that this method throws an **IOException**, we will be forced to handle that exception from the place where we call this **loadFrom()** method. Also notice that we are calling the zero-parameter constructor for the **Car** here ... we would need to make sure that such a constructor is available.

Now we will write some test code to see if it works. Notice in the following code how we separated the write and read tests ...

```
import java.io.*;

public class CarSaveLoadTestProgram {
    private static void writeTest() throws IOException {
        PrintWriter file1, file2;
        Car car1, car2;

        file1 = new PrintWriter(new FileWriter("car1.txt"));
        file2 = new PrintWriter(new FileWriter("car2.txt"));
        car1 = new Car("Pontiac", "Grand-Am", "White", 160, true);
        car2 = new Car("Ford", "Mustang", "White", 230, false);
        car1.saveTo(file1);
        car2.saveTo(file2);
        file1.close();
        file2.close();
    }

    private static void readTest() throws IOException {
        BufferedReader file1, file2;
        Car car1, car2;

        file1 = new BufferedReader(new FileReader("car1.txt"));
        file2 = new BufferedReader(new FileReader("car2.txt"));
        car1 = Car.loadFrom(file1);
        car2 = Car.loadFrom(file2);
        System.out.println(car1);
        System.out.println(car2);
        file1.close();
        file2.close();
    }

    public static void main(String[] args) throws IOException {
        writeTest();
        readTest();
    }
}
```

Notice that we simply ignored handling any **IOExceptions** by declaring that the test methods and the **main()** method all throw the **IOException**. If we now look at the **"car1.txt"** and **"car2.txt"** files, we see that it seems to have saved properly:

car1.txt looks like this:

```
Pontiac
Grand-Am
White
160
true
```

car2.txt looks like this:

```
Ford
Mustang
White
230
false
```

Now for the fun part. Let us make this work with the **Autoshow**. We will make **saveTo()** and **loadFrom()** methods in the **Autoshow** class as well. This time, we need to save ALL the **Car** objects from the autoshow's **cars** list. Recall that we need to first save the autoshow's name to the file:

```
public void saveTo(PrintWriter aFile) {
    aFile.println(name);
    ...
}
```

Now we can iterate through the cars and save them one by one, leaving a blank line in between each, to make the file more readable:

```
public void saveTo(PrintWriter aFile) {
    aFile.println(name);
    for (Car c: cars) {
        aFile.println(); // Leave a blank line before writing the next one
        c.saveTo(aFile);
    }
}
```

Notice that we are making use of the **Car** class's **saveTo()** method. This is important, since it makes good use of pre-existing code and is more modular. Again, we should **import java.io.*** at the top of our **Autoshow** class.

The method for loading an **Autoshow** from the file is also quite easy. It should create and return an **Autoshow** object whose name is the name that is the first line of the file.

We will make it a **static** method as well:

```
public static Autoshow loadFrom(BufferedReader aFile) throws IOException {
    Autoshow aShow = new Autoshow(aFile.readLine());
    ...
    return aShow;
}
```

Again, we will need to make sure that a zero-parameter constructor is available in the **Autoshow** class. Now we now need to read in the name at the top of the file and then read in each car individually. How do we know how many cars to read? Well, we can simply read until there are no more cars left. The **ready()** method in the **BufferedReader** class returns **true** as long as there is another line to be read in the file, otherwise it returns **false**. We can simply keep reading in cars until we get a **!ready()** as follows ...

```

public static Autoshow loadFrom(BufferedReader aFile) throws IOException {
    Autoshow aShow = new Autoshow(aFile.readLine());

    while (aFile.ready()) { //read until no more available (i.e., not ready)
        aFile.readLine(); //read the blank line
        aShow.cars.add(Car.loadFrom(aFile)); //read & add the car
    }
    return aShow;
}

```

Notice that each time we load a new car from the file, we must not forget to add it to the new autoshow object's **cars** collection. Here is a method for testing out our saving and loading:

```

import java.io.*;
public class AutoshowSaveLoadTestProgram {
    // This method tests the writing of an autoshow to a file
    private static void writeTest() throws IOException {
        // First make an Autoshow and add lots of cars to the show
        Autoshow show = new Autoshow("AutoRama 2009");
        show.addCar(new Car("Porsche", "959", "Red", 240, false));
        show.addCar(new Car("Pontiac", "Grand-Am", "White", 160, true));
        show.addCar(new Car("Ford", "Mustang", "White", 230, false));
        show.addCar(new Car("Volkswagon", "Beetle", "Blue", 140, false));
        show.addCar(new Car("Volkswagon", "Jetta", "Silver", 180, true));
        show.addCar(new Car("Geo", "Storm", "Yellow", 110, true));
        show.addCar(new Car("Toyota", "MR2", "Black", 220, false));
        show.addCar(new Car("Ford", "Escort", "Yellow", 10, true));
        show.addCar(new Car("Honda", "Civic", "Black", 220, true));
        show.addCar(new Car("Nissan", "Altima", "Silver", 180, true));
        show.addCar(new Car("BMW", "5", "Gold", 260, true));
        show.addCar(new Car("Prelude", "Honda", "White", 90, false));
        show.addCar(new Car("Mazda", "RX7", "Red", 240, false));
        show.addCar(new Car("Mazda", "MX6", "Green", 160, true));
        show.addCar(new Car("Pontiac", "G6", "Black", 140, false));

        // Now open the file and save the autoshow
        PrintWriter aFile;
        aFile = new PrintWriter(new FileWriter("autoshow.txt"));
        show.saveTo(aFile);
        aFile.close();
    }

    // This method tests the reading of an autoshow from a file
    private static void readTest() throws IOException {
        BufferedReader aFile;

        aFile = new BufferedReader(new FileReader("autoshow.txt"));
        Autoshow aShow = Autoshow.loadFrom(aFile);
        aShow.showCars();
        aFile.close();
    }

    public static void main(String[] args) throws IOException {
        writeTest(); // Write an autoshow to the file
        readTest(); // Read an autoshow from the file
    }
}

```

From running the test, we can see that the **Autoshow** does indeed load properly.

Storing Data on a Single Line:

What if we want to store the **Car** data on a single line as follows:

```
Porsche 959 Red 240 false
Pontiac Grand-Am White 160 true
Ford Mustang White 230 false
```

Well, saving a **Car** is easy:

```
public void saveTo(PrintWriter aFile) {
    aFile.println(make + " " + model + " " + color + " " +
        topSpeed + " " + has4Doors);
}
```

For reading however, we will need to alter the load method to use a **StringTokenizer** to extract the pieces:

```
public static Car loadFrom(BufferedReader aFile) throws IOException {
    Car loadedCar = new Car();

    StringTokenizer wholeLine = new StringTokenizer(aFile.readLine());
    loadedCar.make = wholeLine.nextToken();
    loadedCar.model = wholeLine.nextToken();
    loadedCar.color = wholeLine.nextToken();
    loadedCar.topSpeed = Integer.parseInt(wholeLine.nextToken());
    loadedCar.has4Doors = Boolean.parseBoolean(wholeLine.nextToken());

    return loadedCar;
}
```

The methods for saving and loading the **Autoshow** would not change much, except that the blank line need not be written nor read in after each **Car**.

But what if the **Car** make has two words like this ?

```
PT Cruiser Chrysler Silver 120 true
```

Now it is tougher since the name requires two tokens, not one. We can save and load using commas as our delimiters and then extract the pieces of data one at a time ...

```
PT Cruiser,Chrysler,Silver,120,true
```

This solves the problem.

11.6 The File Class

The **File** class allows us to retrieve information about a file or folder on our computer. However, it has nothing to do with reading and writing information to and from the files.

To make a **File** object, there are three commonly used constructors:






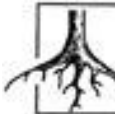
```
File file1 = new File("C:\\Data\\myFile.dat");
File file2 = new File("C:\\Data\\", "myFile.dat");
File file3 = new File(new File("."), "myFile.dat");
```







In the first constructor, we supply the entire file name as a string which includes the path to the file. (A **path** is a sequence of folders on the computer that lead to the file ... starting with the root drive letter).

In the second constructor, we can supply the pathname as a separate string from the file name. The third constructor actually uses another **File** object as a parameter which must represent a folder/directory on the computer. The “.” as a filename indicates the current directory/folder. The “..” as a filename indicates the directory/folder above the current directory/folder. Alternatively we could supply any path name here.




Once we create this **File** object, there are a set of methods that we can use to ask questions about this file or folder. Here are just some of the available methods:

boolean canRead() Returns whether or not this file is readable.	
boolean canWrite() Returns whether or not this file is writable.	
boolean exists() Returns whether or not this file or directory exists in the specified path.	
boolean isFile() Returns whether or not this represents a file (as opposed to a directory/folder).	
boolean isDirectory() Returns whether or not this represents a directory/folder (as opposed to a file).	
boolean isAbsolute() Returns whether or not this represents an absolute path to a file or directory.	

Here are other methods for accessing components (i.e., filenames and pathnames) of a file or directory:

String getAbsolutePath() Return a String with the absolute path of the file or directory.	
String getName() Return a String with the name of the file or directory.	
String getPath() Return a String with the path of the file or directory.	
String getParent() Return a String with the parent directory of the file or directory.	

Here are some other user useful methods:

long length() Return the length of the file in bytes. If the File object is a directory, return 0.	
long lastModified() Return a system-dependent representation of the time at which the file or directory was last modified. The value returned is only useful for comparison with other values returned by this method.	
String[] list() Return an array of Strings representing the contents of a directory.	

For the purpose of demonstration, here is a program that gives a directory listing of the files and folders on the root of your **C:** drive, but it does not go into each folder recursively ...

```
import java.io.*;

public class FileClassTestProgram {
    public static void main(String[] args) {
        // The dot means the current directory
        File currDir = new File("C:\\");
        System.out.println("The directory name is: " + currDir.getName());
        System.out.println("The path name is: " + currDir.getPath());
        System.out.println("The actual path name is: " +
            currDir.getAbsolutePath());

        System.out.println("Here are the files in the current directory: ");
        String[] files = currDir.list();
        for (int i=0; i<files.length; i++) {
            if (new File("C:\\", files[i]).isDirectory())
                System.out.print("*** ");
            System.out.println(files[i]);
        }
    }
}
```

Here is the output (which differs of course depending where you run your code from):

```
The directory name is:
The path name is: C:\
The actual path name is: C:\
Here are the files in the current directory:
*** 1eceb6cd306883b5737c1dbf5404e4
a-1049-1-7C23.zip
AUTOEXEC.BAT
BOOT.BAK
boot.ini
*** Config.Msi
CONFIG.SYS
*** CtDriverInstTemp
*** Documents and Settings
*** Downloaded Videos
*** Downloads
*** drivers
hiberfil.sys
*** i386
INFCACHE.1
IO.SYS
IPH.PH
*** java

... etc ...

*** System Volume Information
*** temp
*** TempArchive
*** Users
*** WINDOWS
```

File Separators:

Depending on which type of computer you have, folders are specified in different ways. For example, windows uses a ‘\’ character to separate folders in a pathname, whereas Unix/Linux uses ‘/’ and Classic Mac OS uses “.”.

If we were to hard-code out pathnames into our programs, then our code would not be portable to different machines. For example, this pathname:

```
String fileName = "C:\\FunInc\\models\\BankAccount.java";
```

would be ok for a windows-based machine, but for a Unix/Linux machine, the pathname would be invalid. We would need to use something like this for Linux:

```
String fileName = "usr/FunInc/models/BankAccount.java";
```

... and further...something like this for an older Mac OS:

```
String fileName = "C:FunInc:models:BankAccount.java";
```

In order to make our code portable, JAVA has defined a **static** constant called **separator** in the **File** class which will represent the appropriate file separation character depending on the machine that our code is running on. Hence, the following code will be portable for all machines:

```
String fileName = File.separator + "FunInc" + File.separator +  
                  "models" + File.separator + "BankAccount.java";
```

If you do this in your programs, your code will always be portable and you will save time when porting your code to other machines.

This page has been intentionally left blank.

Chapter 12

Network Programming

What is in This Chapter ?

This chapter explains how to connect your JAVA application to a network. You will learn how to read files from over the internet as well as have two or more programs communicate with one another over a network connection (wired or wireless). You will learn about **Uniform Resource Locators** as well as **Client/Server** communications using **TCP** and **Datagram Sockets**.



12.1 Networking Basics

Network Programming involves writing programs that communicate with other programs across a computer network.

There are many issues that arise when doing network programming which do not appear when doing single program applications. However, JAVA makes networking applications simple due to the easy-to-use libraries. In general, applications that have components running on different machines are known as **distributed** applications ... and usually they consist of client/server relationships.

A **server** is an application that provides a "service" to various **clients** who request the service.

There are many client/server scenarios in real life:

- Bank tellers (server) provide a service for the account owners (client)
- Waitresses (server) provide a service for customers (client)
- Travel agents (server) provide a service for people wishing to go on vacation (client)



In some cases, servers themselves may become clients at various times.

- E.g., travel agents will become clients when they phone the airline to make a reservation or contact a hotel to book a room.

In the general networking scenario, everybody can either be a client or a server at any time. This is known as **peer-to-peer** computing. In terms of writing java applications it is similar to having many applications communicating among one another.

- E.g., the original **Napster** worked this way. Thousands of people all acted as clients (trying to download songs from another person) as well as servers (in that they allowed others to download their songs).



There are many different strategies for allowing communication between applications. JAVA technology allows:

- internet clients to connect to servlets or back-end business systems (or databases).
- applications to connect to one another using sockets.
- applications to connect to one another using RMI (remote method invocation).
- some others

We will look at the simplest strategy of connecting applications using sockets.

A **Protocol** is a standard pattern of exchanging information.

It is like a set of rules/steps for communication. The simplest example of a protocol is a phone conversation:

1. **JIM** dials a phone number
2. **MARY** says "Hello..."
3. **JIM** says "Hello..."
4. The conversation goes on for a while ...
5. **JIM** says "Goodbye"
6. **MARY** says "Goodbye"



Perhaps another person gets involved:

1. **JIM** dials a phone number
2. **MARY** says "Hello..."
3. **JIM** says "Hello" and perhaps asks to speak to **FRED**
4. **MARY** says "Just a minute"
5. **FRED** says "Hello..."
6. **JIM** says "Hello..."
7. The conversation goes on for a while ...
8. **JIM** says "Goodbye"
9. **FRED** says "Goodbye"

Either way, there is an "expected" set of steps or responses involved during the initiation and conclusion of the conversation. If these steps are not followed, confusion occurs (like when you phone someone and they pick up the phone but do not say anything).

Computer protocols are similar in that a certain amount of "*handshaking*" goes on to establish a valid connection between two machines. Just as we know that there are different ways to shake hands, there are also different protocols. There are actually layered levels of protocols in that some low level layers deal with how to transfer the data bits, others deal with more higher-level issues such as "where to send the data to".

Computers running on the internet typically use one of the following high-level **Application Layer** protocols to allow applications to communicate:

- **Hyper Text Transfer Protocol (HTTP)**
- **File Transfer Protocol (FTP)**
- **Telnet**

This is analogous to having multiple strategies for communicating with someone (in person, by phone, through electronic means, by post office mail etc...).

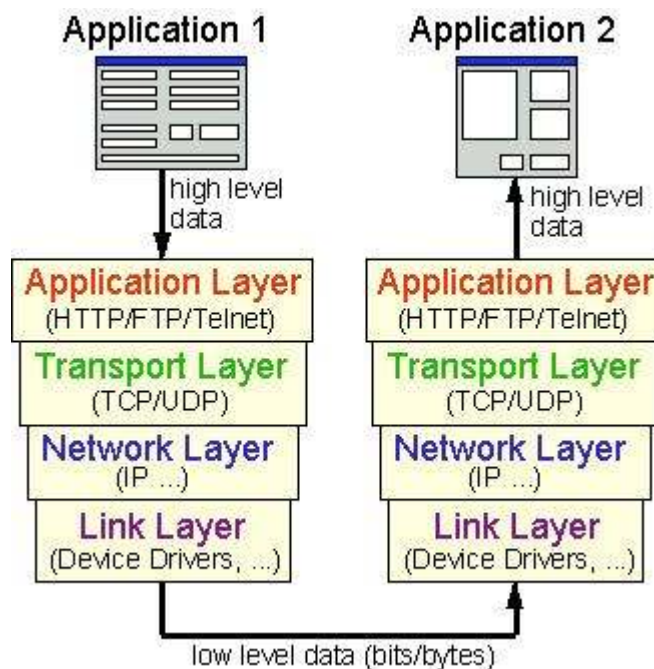
In a lower **Transport Layer** of communication, there is a separate protocol which is used to determine how the data is to be transported from one machine to another:

- **Transport Control Protocol (TCP)**
- **User Datagram Protocol (UDP)**

This is analogous to having multiple ways of actually delivering a package to someone (Email, Fax, UPS, Fed-Ex etc...)

Beneath that layer is a **Network Layer** for determining how to locate destinations for the data (i.e., address). And at the lowest level (for computers) there is a **Link Layer** which actually handles the transferring of bits/bytes.

So, internet communication is built of several layers:

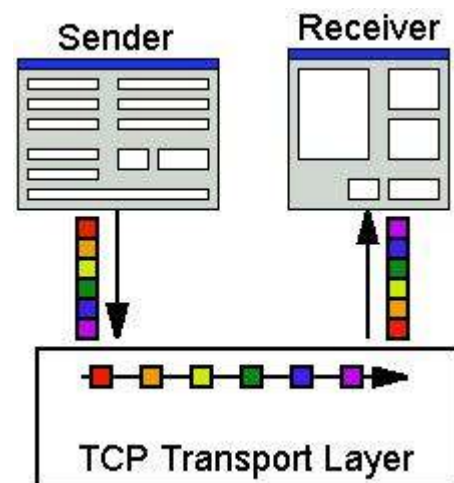


When you write JAVA applications that communicate over a network, you are programming in the **Application Layer**.

JAVA allows two types of communication via two main types of **Transport Layer** protocols:

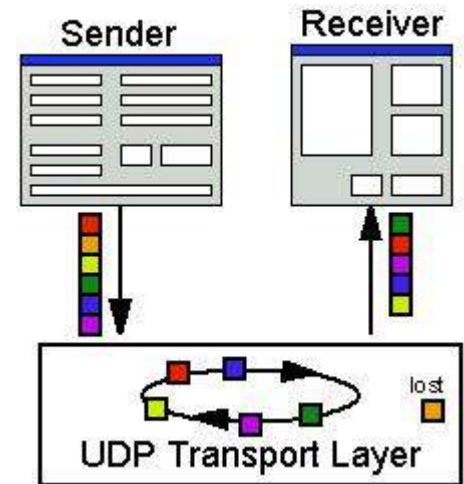
TCP

- a **connection-based** protocol that provides a reliable flow of data between two computers.
- guarantees that data sent from one end of the connection actually gets to the other end and in the same order
 - similar to a phone call. Your words come out in the order that you say them.
- provides a point-to-point channel for applications that require **reliable communications**.
- **slow overhead time** of setting up an end-to-end connection.

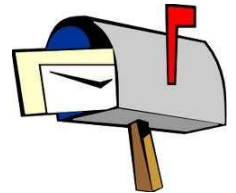


UDP

- a protocol that sends independent packets of data, called **datagrams**, from one computer to another.
- no guarantees about arrival. UDP is not connection-based like TCP.
- provides **communication that is not guaranteed** between the two ends
 - sending packets is like sending a letter through the postal service
 - the order of delivery is not important and not guaranteed
 - each message is independent of any other
- **faster** since no overhead of setting up end-to-end connection
- many firewalls and routers have been configured **NOT TO** allow UDP packets.



Why would anyone want to use UDP protocol if information may get lost ? Well, why do we use email or the post office ? We are never guaranteed that our mail will make it to the person that we send it to, yet we still rely on those delivery services. It may still be quicker than trying to contact a person via phone to convey the data (i.e., like a TCP protocol).



One more important definition we need to understand is that of a *port*:

*A **port** is used as a gateway or "entry point" into an application.*

Although a computer usually has a single physical connection to the network, data sent by different applications or delivered to them do so through the use of ports configured on the same physical network connection. Data transmitted over the internet to an application requires the address of the destination computer and the application's port number. A computer's address is a 32-bit IP address. The port number is a 16-bit number ranging from 0 to 65,535, with ports 0-1023 restricted by well-known applications like HTTP and FTP.

12.2 Reading Files From the Internet (URLs)

A **Uniform Resource Locator** (i.e., **URL**) is a reference (or address) to a resource over a network (e.g., on the Internet).

So, a URL can be used to represent the "location" of a webpage or web-based application. A URL is really just a String that represent the names of resources which can be files, databases, applications, etc.. The resource names contain a host machine name, filename, port number, and other information and may also specify a **protocol identifier** (e.g., http, ftp) Here are some examples of URLs:

```
http://www.cnn.com/  
http://www.apple.com/ipad/index.html  
http://en.wikipedia.org/wiki/Computer_science
```

Here, **http://** is the protocol identifier which indicates the protocol that will be used to obtain the resource. The remaining part is the **resource name**, and its format depends on the protocol used to access it.

The complete list of components that can be found in a URL resource name are as follows:

- **Host Name** - The name of the machine on which the resource lives.
`http://www.apple.com:80/ipad/index.html`
- **Port # (optional)** - The port number to which to connect.
`http://www.apple.com:80/ipad/index.html`
- **Filename** - The pathname to the file on the machine.
`http://www.apple.com:80/ipad/index.html`

In JAVA, there is a **URL** class defined in the **java.net** package. We can create our own URL objects as follows:

```
URL webPage = new URL("http://www.apple.com/ipad/index.html");
```

JAVA will "dissect" the given String in order to obtain information about protocol, hostName, file etc....

Due to this, JAVA may throw a **MalformedURLException** ... so we will need to do this:

```
try {  
    URL webPage = new URL("http://www.apple.com/ipad/index.html");  
} catch (MalformedURLException e) {  
    ...  
}
```

Another way to create a URL is to break it into its various components:

```
try {
    URL webPage = new URL("http", "www.apple.com", 80, "/ipad/index.html");
} catch (MalformedURLException e) {
    ...
}
```

If you take a look at the JAVA API, you will notice some other constructors as well.

The URL class also supplies methods for extracting the parts (protocol, host, file, port and reference) of a URL object. Here is an example that demonstrates what can be accessed. Note that this example only manipulates a URL object, it does not go off to grab any web pages:

```
import java.net.*;

public class URLTestProgram {
    public static void main(String[] args) {
        URL webpage = null;
        try {
            webpage = new URL("http", "www.apple.com", 80, "/ipad/index.html");
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        System.out.println(webpage);
        System.out.println("protocol = " + webpage.getProtocol());
        System.out.println("host = " + webpage.getHost());
        System.out.println("filename = " + webpage.getFile());
        System.out.println("port = " + webpage.getPort());
        System.out.println("ref = " + webpage.getRef());
    }
}
```

Here is the output:

```
http://www.apple.com:80/ipad/index.html
protocol = http
host = www.apple.com
filename = /ipad/index.html
port = 80
ref = null
```

After creating a URL object, you can actually connect to that webpage and read the contents of the URL by using its **openStream()** method which returns an **InputStream**. You actually read from the webpage as if it were a simple text file. If an attempt is made to read from a URL that does not exist, JAVA will throw an **UnknownHostException**

Example:

Here is an example that reads a URL directly. It actually reads the file on wikipedia and displays it line by line to the console. Notice that it reads the file as a text file, so we simply get the HTML code. Also, you must be connected to the internet to run this code:

```
import java.net.*;
import java.io.*;

public class URLReaderProgram {
    public static void main(String[] args) {
        URL wiki = null;
        try {
            wiki = new URL("http", "en.wikipedia.org", 80,
                "/wiki/Computer_science");
            BufferedReader in = new BufferedReader(
                new InputStreamReader(wiki.openStream()));

            // Now read the webpage file
            String lineOfWebPage;
            while ((lineOfWebPage = in.readLine()) != null)
                System.out.println(lineOfWebPage);

            in.close(); // Close the connection to the net
        } catch (MalformedURLException e) {
            System.out.println("Cannot find webpage " + wiki);
        } catch (IOException e) {
            System.out.println("Cannot read from webpage " + wiki);
        }
    }
}
```

The output should look something like this, assuming you could connect to the webpage:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="en" dir="ltr" xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Computer science - Wikipedia, the free encyclopedia</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<meta http-equiv="Content-Style-Type" content="text/css" />
<meta name="generator" content="MediaWiki 1.17wmf1" />
<link rel="alternate" type="application/x-wiki" title="Edit this page"
href="/w/index.php?title=Computer_science&action=edit" />
<link rel="edit" title="Edit this page"
href="/w/index.php?title=Computer_science&action=edit" />
<link rel="apple-touch-icon" href="http://en.wikipedia.org/apple-touch-icon.png" />
<link rel="shortcut icon" href="/favicon.ico" />
<link rel="search" type="application/opensearchdescription+xml"
href="/w/opensearch_desc.php" title="Wikipedia (en)" />
...
```

Example:

Here is a modification to the above example that reads the URL by making a **URLConnection** first. Since the tasks of opening a connection to a webpage and reading the contents may both generate an **IOException**, we cannot distinguish the kind of error that occurred. By trying to establish the connection first, if any **IOExceptions** occur, we know they are due to a connection problem. Once the connection has been established, then any further **IOException** errors would be due to the reading of the webpage data.

```
import java.net.*;
import java.io.*;

public class URLConnectionReaderExample {
    public static void main(String[] args) {
        URL wiki = null;
        BufferedReader in = null;
        try {
            wiki = new URL("http", "en.wikipedia.org", 80,
                "/wiki/Computer_science");
        } catch (MalformedURLException e) {
            System.out.println("Cannot find webpage " + wiki);
            System.exit(-1);
        }
        try {
            URLConnection aConnection = wiki.openConnection();
            in = new BufferedReader(
                new InputStreamReader(aConnection.getInputStream()));
        }
        catch (IOException e) {
            System.out.println("Cannot connect to webpage " + wiki);
            System.exit(-1);
        }
        try {
            // Now read the webpage file
            String lineOfWebPage;
            while ((lineOfWebPage = in.readLine()) != null)
                System.out.println(lineOfWebPage);
            in.close(); // Close the connection to the net
        } catch (IOException e) {
            System.out.println("Cannot read from webpage " + wiki);
        }
    }
}
```

12.3 Client/Server Communications

Many companies today sell services or products. In addition, there are a large number of companies turning towards E-business solutions and various kinds of web-server/database technologies that allow them to conduct business over the internet as well as over other networks. Such applications usually represent a client/server scenario in which one or more servers serve multiple clients.

A **server** is any application that provides a service and allows clients to communicate with it.



Such services may provide:

- a recent stock quote
- transactions for bank accounts
- an ability to order products
- an ability to make reservations
- a way to allow multiple clients to interact (Auction)

A **client** is any application that requests a service from a server.

The client typically "uses" the service and then displays results to the user. Normally, communication between the client and server must be reliable (no data can be dropped or missing):

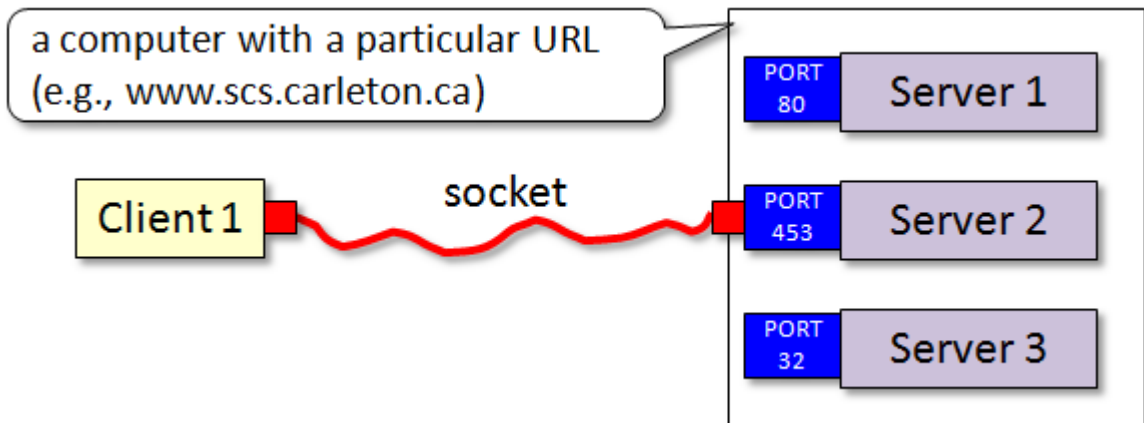


- stock quotes must be accurate and timely
- banking transactions must be accurate and stable
- reservations/orders must be acknowledged

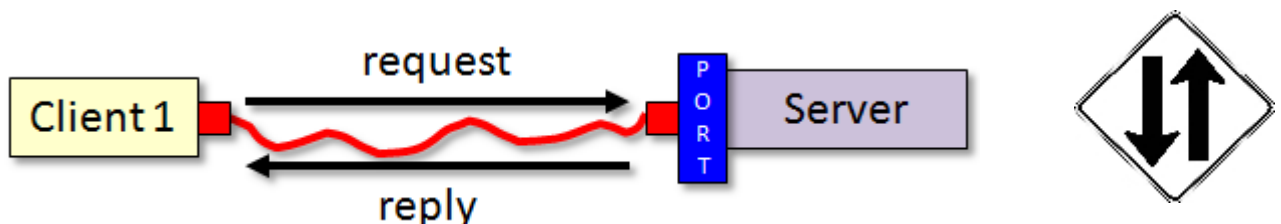
The TCP protocol, mentioned earlier, provides reliable point-to-point communication. Using TCP the client and server must establish a connection in order to communicate. To do this, each program binds a **socket** to its end of the connection. A **socket** is one endpoint of a two-way communication link between 2 programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application to which the data is to be sent. It is similar to the idea of plugging the two together with a cable.



The **port number** is used as the server's location on the machine that the server application is running. So if a computer is running many different server applications on the same physical machine, the port number uniquely identifies the particular server that the client wishes to communicate with:



The client and server may then each read and write to the socket bound to its end of the connection.



In JAVA, the server application uses a **ServerSocket** object to wait for client connection requests. When you create a **ServerSocket**, you must specify a port number (an **int**). It is possible that the server cannot set up a socket and so we have to expect a possible **IOException**. Here is an example:

```
public static int SERVER_PORT = 5000;

ServerSocket serverSocket;
try {
    serverSocket = new ServerSocket(SERVER_PORT);
} catch (IOException e) {
    JOptionPane.showMessageDialog(null, "Cannot open server connection",
        "Error", JOptionPane.ERROR_MESSAGE);
}
```

The server can communicate with only one client at a time.

The server waits for an incoming client request through the use of the **accept()** message:

```

Socket aClientSocket;
try {
    aClientSocket = serverSocket.accept();
} catch (IOException e) {
    JOptionPane.showMessageDialog(null, "Cannot connect to client",
        "Error", JOptionPane.ERROR_MESSAGE);
}

```

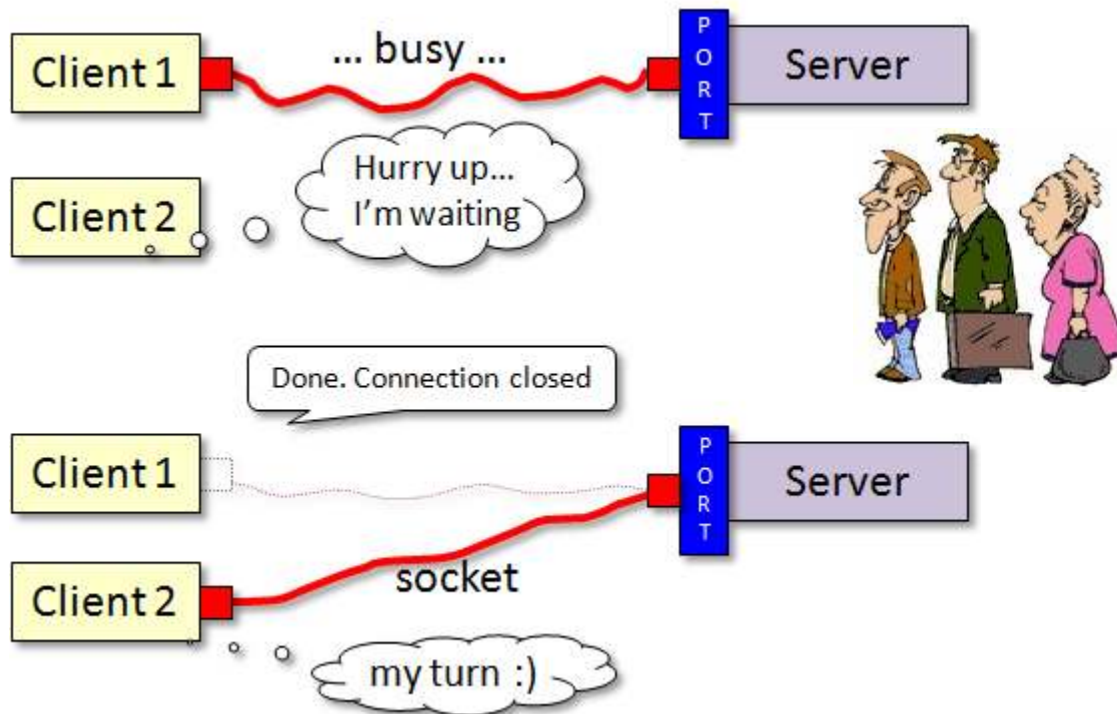
When the **accept()** method is called, the server program actually waits (i.e., **blocks**) until a client becomes available (i.e., an incoming client request arrives). Then it creates and returns a **Socket** object through which communication takes place.



Once the client and server have completed their interaction, the socket is then closed:

```
aClientSocket.close();
```

Only then may the next client open a socket connection to the server. So, remember ... if one client has a connection, everybody else has to wait until they are done:



So how does the client connect to the server? Well, the client must know the address of the server as well as the port number. The server's address is stored as an **InetAddress** object which represents any IP address (i.e., an internet address, an ftp site, local machine etc,...).

If the server and client are on the same machine, the static method **getLocalHost()** in the **InetAddress** class may be used to get an address representing the local machine as follows:

```

public static int SERVER_PORT = 5000;

try {
    InetAddress address = InetAddress.getLocalHost();
    Socket socket = new Socket(address, SERVER_PORT);
} catch (UnknownHostException e) {
    JOptionPane.showMessageDialog(null, "Host Unknown",
        "Error", JOptionPane.ERROR_MESSAGE);
} catch (IOException e) {
    JOptionPane.showMessageDialog(null, "Cannot connect to server",
        "Error", JOptionPane.ERROR_MESSAGE);
}

```

Once again, a socket object is returned which can then be used for communication. Here is an example of what a local host may look like:

```
cr850205-a/169.254.180.32
```

The `getLocalHost()` method may, however, generate an **UnknownHostException**. You can also make an **InetAddress** object by specifying the network IP address directly or the machine name directly as follows:

```

InetAddress.getByName("169.254.1.61");
InetAddress.getByName("www.scs.carleton.ca");

```

So how do we actually do communication between the client and the server? Well, each socket has an **InputStream** and an **OutputStream**. So, once we have the sockets, we simply ask for these streams ... and then reading and writing may occur.

```

try {
    InputStream in = socket.getInputStream();
    OutputStream out = socket.getOutputStream();
} catch (IOException e) {
    JOptionPane.showMessageDialog(null, "Cannot open I/O Streams",
        "Error", JOptionPane.ERROR_MESSAGE);
}

```

Normally, however, we actually wrap these input/output streams with text-based, datatype-based or object-based wrappers:

```

ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

```

```

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
PrintWriter out = new PrintWriter(socket.getOutputStream());

```

```

DataInputStream in = new DataInputStream(socket.getInputStream());
DataOutputStream out = new DataOutputStream(socket.getOutputStream());

```

You may look back at the notes on streams to see how to write to the streams. However, one more point ... when data is sent through the output stream, the **flush()** method should be sent to the output stream so that the data is not buffered, but actually sent right away.

Also, you must be careful when using **ObjectInputStreams** and **ObjectOutputStreams**. When you create an **ObjectInputStream**, it blocks while it tries to read a header from the underlying **SocketInputStream**. When you create the corresponding **ObjectOutputStream** at the far end, it writes the header that the **ObjectInputStream** is waiting for, and both are able to continue. If you try to create both **ObjectInputStreams** first, each end of the connection is waiting for the other to complete before proceeding which results in a deadlock situation (i.e., the programs seems to hang/halt). This behavior is described in the API documentation for the **ObjectInputStream** and **ObjectOutoutStream** constructors.

Example:

Lets now take a look at a real example. In this example, a client will attempt to:

1. connect to a server
2. ask the server for the current time
3. ask the server for the number of requests that the server has handled so far
4. ask the server for an invalid request (i.e., for a pizza)

Here is the server application. It runs forever, continually waiting for incoming client requests:

```
import java.net.*; // all socket stuff is in here
import java.io.*;
import javax.swing.JOptionPane;

public class Server {
    public static int SERVER_PORT = 6000; // arbitrary, but above 1023
    private int counter = 0;

    // Helper method to get the ServerSocket started
    private ServerSocket goOnline() {
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(SERVER_PORT);
        } catch (IOException e) {
            JOptionPane.showMessageDialog(null,
                "SERVER: Error creating network connection",
                "Error", JOptionPane.ERROR_MESSAGE);
        }
        System.out.println("SERVER online");
        return serverSocket;
    }

    // Handle all requests
    private void handleRequests(ServerSocket serverSocket) {

        while(true) {
            Socket socket = null;
            BufferedReader in = null;
            PrintWriter out = null;

            try {
```

```

        // Wait for an incoming client request
        socket = serverSocket.accept();
        // At this point, a client connection has been made
        in = new BufferedReader(new InputStreamReader(
                                socket.getInputStream()));
        out = new PrintWriter(socket.getOutputStream());
    } catch(IOException e) {
        JOptionPane.showMessageDialog(null,
                                    "SERVER: Error connecting to client",
                                    "Error", JOptionPane.ERROR_MESSAGE);

        System.exit(-1);
    }
    // Read in the client's request
    try {
        String request = in.readLine();
        System.out.println("SERVER: Client Message Received: " + request);
        if (request.equals("What Time is It ?")) {
            out.println(new java.util.Date());
            counter++;
        }
        else if (request.equals("How many requests have you handled ?"))
            out.println(counter++);
        else
            System.out.println("SERVER: Unknown request: " + request);

        out.flush();           // Now make sure that the response is sent
        socket.close();        // We are done with the client's request

    } catch(IOException e) {
        JOptionPane.showMessageDialog(null,
                                    "SERVER: Error communicating with client",
                                    "Error", JOptionPane.ERROR_MESSAGE);
    }
}

public static void main (String[] args) {
    Server s = new Server ();
    ServerSocket ss = s.goOnline();
    if (s != null)
        s.handleRequests(ss);
}
}

```

Here is the client application:

```

import java.net.*;
import java.io.*;
import javax.swing.JOptionPane;

public class ClientProgram {
    private Socket      socket;
    private BufferedReader in;
    private PrintWriter out;
}

```

```
// Make a connection to the server
private void connectToServer() {
    try {
        socket = new Socket(InetAddress.getLocalHost(), Server.SERVER_PORT);

        in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        out = new PrintWriter(socket.getOutputStream());
    } catch (IOException e) {
        JOptionPane.showMessageDialog(null, "CLIENT: Cannot connect to server",
            "Error", JOptionPane.ERROR_MESSAGE);

        System.exit(-1);
    }
}

// Disconnect from the server
private void disconnectFromServer() {
    try {
        socket.close();
    } catch (IOException e) {
        JOptionPane.showMessageDialog(null,
            "CLIENT: Cannot disconnect from server",
            "Error", JOptionPane.ERROR_MESSAGE);
    }
}

// Ask the server for the current time
private void askForTime() {
    connectToServer();
    out.println("What Time is It ?");
    out.flush();
    try {
        String time = in.readLine();
        System.out.println("CLIENT: The time is " + time);
    } catch (IOException e) {
        JOptionPane.showMessageDialog(null,
            "CLIENT: Cannot receive time from server",
            "Error", JOptionPane.ERROR_MESSAGE);
    }
    disconnectFromServer();
}

// Ask the server for the number of requests obtained
private void askForNumberOfRequests() {
    connectToServer();
    out.println("How many requests have you handled ?");
    out.flush();
    int count = 0;
    try {
        count = Integer.parseInt(in.readLine());
    } catch (IOException e) {
        JOptionPane.showMessageDialog(null,
            "CLIENT: Cannot receive num requests from server",
            "Error", JOptionPane.ERROR_MESSAGE);
    }
    System.out.println("CLIENT: The number of requests are " + count);
    disconnectFromServer();
}
```

```

// Ask the server to order a pizza
private void askForAPizza() {
    connectToServer();
    out.println("Give me a pizza");
    out.flush();
    disconnectFromServer();
}

public static void main (String[] args) {
    ClientProgram c = new ClientProgram();
    c.askForTime();
    c.askForNumberOfRequests();
    c.askForAPizza();
    c.askForTime();
    c.askForNumberOfRequests();
}
}

```

Note, to run this using JCreator, we will have to execute two different JCreator applications, one for the server and one for the client.

12.4 Datagram Sockets

Recall that with the datagram protocol (i.e., UDP) there is no direct socket connection between the client and the server. That is, packets are received "in seemingly random order" from different clients. It is similar to the way email works. If the client requests or server responses are too big, they are broken up into multiple packets and sent one packet at a time. The server is not guaranteed to receive the packets all at once, nor in the same order, nor is it guaranteed to receive all the packets !!

Let us look at the same client-server application, but by now using **DatagramSockets** and **DatagramPackets**. Once again, the server will be in a infinite loop accepting messages, although there will be no direct socket connection to the client. We will be setting up a **buffer** (i.e., an array of bytes) which will be used to receive incoming requests. Each message is sent as a **packet**. Each packet contains:

- the **data** of message (i.e., the message itself)
- the **length** of the message (i.e., the number of bytes)
- the **address** of the sender (as an `InetAddress`)
- the **port** of the sender



The code for packaging and sending an outgoing packet involves creating a **DatagramSocket** and then constructing a **DatagramPacket**. The packet requires an array of bytes, as well as the address and port in which to send to. The byte array can be obtained from most objects by sending a **getBytes()** message to the object. Finally, a **send()** message is used to send the packet:

```

byte[]          sendBuffer;
DatagramSocket socket;
DatagramPacket packetToSend ;

socket = new DatagramSocket();
sendBuffer = "This is the data ... need not be a String".getBytes();
packetToSend = new DatagramPacket(sendBuffer, sendBuffer.length,
                                   anInetAddress, aPort);

socket.send(packetToSend);

```

The server code for receiving an incoming packet involves allocating space (i.e., a byte array) for the **DatagramPacket** and then receiving it. The code looks as follows:

```

byte[]          recieveBuffer;
DatagramPacket receivePacket;

recieveBuffer = new byte[INPUT_BUFFER_LIMIT];
receivePacket = new DatagramPacket(recieveBuffer, recieveBuffer.length);
socket.receive(receivePacket);

```

We then need to extract the data from the packet. We can get the address and port of the sender as well as the data itself from the packet as follows:

```

InetAddress sendersAddress = receivePacket.getAddress();
int          sendersPort   = receivePacket.getPort();
String       sendersData   = new String(receivePacket.getData(), 0,
                                       receivePacket.getLength());

```

In this case the data sent was a **String**, although it may in general be any object. By using the sender's address and port, whoever receives the packet can send back a reply.

Example:

Here is a modified version of our client/server code ... now using the **DatagramPackets**:

```

import java.net.*;
import java.io.*;
import javax.swing.JOptionPane;

public class PacketServer {
    public static int SERVER_PORT = 6000;
    private static int INPUT_BUFFER_LIMIT = 500;
    private int counter = 0;

    // Handle all requests
    private void handleRequests() {
        System.out.println("SERVER online");

        // Create a socket for communication
        DatagramSocket socket = null;
        try {
            socket = new DatagramSocket(SERVER_PORT);

```



```

    } catch (SocketException e) {
        JOptionPane.showMessageDialog(null, "SERVER: no network connection",
            "Error", JOptionPane.ERROR_MESSAGE);

        System.exit(-1);
    }
    // Now handle incoming requests
    while(true) {
        try {
            // Wait for an incoming client request
            byte[] recieveBuffer = new byte[INPUT_BUFFER_LIMIT];
            DatagramPacket receivePacket;
            receivePacket = new DatagramPacket(recieveBuffer,
                recieveBuffer.length);

            socket.receive(receivePacket);
            // Extract the packet data that contains the request
            InetAddress address = receivePacket.getAddress();
            int clientPort = receivePacket.getPort();
            String request = new String(receivePacket.getData(), 0,
                receivePacket.getLength());

            System.out.println("SERVER: Packet received: \"" + request +
                "\" from " + address + ":" + clientPort);

            // Decide what should be sent back to the client
            byte[] sendBuffer;
            if (request.equals("What Time is It?")) {
                System.out.println("SERVER: sending packet with time info");
                sendResponse(socket, address, clientPort,
                    new java.util.Date().toString().getBytes());

                counter++;
            }
            else if (request.equals("How many requests have you handled?")) {
                System.out.println("SERVER: sending packet with num requests");
                sendResponse(socket, address, clientPort,
                    ("\" + ++counter).getBytes());
            }
            else
                System.out.println("SERVER: Unknown request: " + request);
        } catch (IOException e) {
            JOptionPane.showMessageDialog(null,
                "SERVER: Error receiving client requests",
                "Error", JOptionPane.ERROR_MESSAGE);
        }
    }
}

// This helper method sends a given response back to the client
private void sendResponse(DatagramSocket socket, InetAddress address,
    int clientPort, byte[] response) {
    try {
        // Now create a packet to contain the response and send it
        DatagramPacket sendPacket = new DatagramPacket(response,
            response.length, address, clientPort);

        socket.send(sendPacket);
    } catch (IOException e) {
        JOptionPane.showMessageDialog(null,
            "SERVER: Error sending response to client" +
            address + ":" + clientPort,
            "Error", JOptionPane.ERROR_MESSAGE);
    }
}
}

```

```

public static void main (String args[]) {
    new PacketServer().handleRequest();
}
}

```

Notice that only one **DatagramSocket** is used, but that a new **DatagramPacket** object is created for each incoming message. Now let us look at the client:

```

import java.net.*;
import java.io.*;
import javax.swing.JOptionPane;

public class PacketClientProgram {
    private static int INPUT_BUFFER_LIMIT = 500;
    private InetAddress localhost;

    public PacketClientProgram() {
        try {
            localhost = InetAddress.getLocalHost();
        } catch (UnknownHostException e) {
            JOptionPane.showMessageDialog(null,
                "CLIENT: Error connecting to network",
                "Error", JOptionPane.ERROR_MESSAGE);

            System.exit(-1);
        }
    }

    // Ask the server for the current time
    private void askForTime() {
        DatagramSocket socket = null;
        try {
            socket = new DatagramSocket();
            byte[] sendBuffer = "What Time is It ?".getBytes();
            DatagramPacket sendPacket = new DatagramPacket(sendBuffer,
                sendBuffer.length, localhost,
                PacketServerProgram.SERVER_PORT);

            System.out.println("CLIENT: Sending time request to server");
            socket.send(sendPacket);
        } catch (IOException e) {
            JOptionPane.showMessageDialog(null,
                "CLIENT: Error sending time request to server",
                "Error", JOptionPane.ERROR_MESSAGE);
        }
        try {
            byte[] receiveBuffer = new byte[INPUT_BUFFER_LIMIT];
            DatagramPacket receivePacket = new DatagramPacket(receiveBuffer,
                receiveBuffer.length);

            socket.receive(receivePacket);
            System.out.println("CLIENT: The time is " + new String(
                receivePacket.getData(), 0, receivePacket.getLength()));
        } catch (IOException e) {
            JOptionPane.showMessageDialog(null,
                "CLIENT: Cannot receive time from server",
                "Error", JOptionPane.ERROR_MESSAGE);
        }
        socket.close();
    }
}

```

```
// Ask the server for the number of requests obtained
private void askForNumberOfRequests() {
    DatagramSocket socket = null;
    try {
        socket = new DatagramSocket();
        byte[] sendBuffer = "How many requests have you handled?".getBytes();
        DatagramPacket sendPacket = new DatagramPacket(sendBuffer,
                                                       sendBuffer.length, localhost,
                                                       PacketServerProgram.SERVER_PORT);
        System.out.println("CLIENT: Sending request count request to server");
        socket.send(sendPacket);
    } catch (IOException e) {
        JOptionPane.showMessageDialog(null,
                                     "CLIENT: Error sending request to server",
                                     "Error", JOptionPane.ERROR_MESSAGE);
    }
    try {
        byte[] receiveBuffer = new byte[INPUT_BUFFER_LIMIT];
        DatagramPacket receivePacket = new DatagramPacket(receiveBuffer,
                                                         receiveBuffer.length);
        socket.receive(receivePacket);
        System.out.println("CLIENT: The number of requests are " +
                           new String(receivePacket.getData(), 0,
                                       receivePacket.getLength()));
    } catch (IOException e) {
        JOptionPane.showMessageDialog(null,
                                     "CLIENT: Cannot receive num requests from server",
                                     "Error", JOptionPane.ERROR_MESSAGE);
    }
    socket.close();
}

// Ask the server to order a pizza
private void askForAPizza() {
    try {
        byte[] sendBuffer = "Give me a pizza".getBytes();
        DatagramPacket sendPacket = new DatagramPacket(sendBuffer,
                                                       sendBuffer.length, localhost,
                                                       PacketServerProgram.SERVER_PORT);
        DatagramSocket socket = new DatagramSocket();
        System.out.println("CLIENT: Sending pizza request to server");
        socket.send(sendPacket);
        socket.close();
    } catch (IOException e) {
        JOptionPane.showMessageDialog(null,
                                     "CLIENT: Error sending request to server",
                                     "Error", JOptionPane.ERROR_MESSAGE);
    }
}

public static void main (String[] args) {
    PacketClientProgram c = new PacketClientProgram();
    c.askForTime();
    c.askForNumberOfRequests();
    c.askForAPizza();
    c.askForTime();
    c.askForNumberOfRequests();
}
}
```

This page has been intentionally left blank.

Chapter 13

Other Interesting JAVA Classes

What is in This Chapter ?

This chapter discusses some interesting JAVA classes such as **String**, **Date**, **GregorianCalendar** and others. This chapter can be viewed as an explanation of the tool-like classes available in JAVA to make your life easier.



13.1 The String Class

Strings are one of the most commonly used concepts in all programming languages. They are used to represent text characters and are fundamental in allowing a user to interact with the program. In JAVA, Strings are actually objects, not primitives and any text between double quotes represents a *literal* String in our programs:

```
String name = "Stan Dupp";
String empty = "";
```

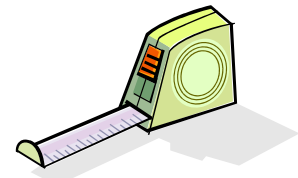
However, since Strings are also objects, we can create one by using one of many available constructors. Here are two examples:

```
String nothing = new String();           // makes an empty String
String copy = new String(name);         // makes copy of the name String
```



A **String** has a *length* corresponding to the number of characters in the String. We can ask a **String** for its length by using the **length()** method:

```
String name = "Stan Dupp";
String empty = "";
name.length();    // returns 9
empty.length();   // returns 0
```



This length remains unchanged for the string at all times. That is, once a string has been created we *cannot change the size* of the string, nor can we *append* to the string.

Even though we cannot append to a String, we can still make use of the **+** operator to join two of them together. Recall, for example, the use of the **+** operator within the **toString()** method for the **Person** class:

```
public String toString() {
    return (this.age + " year old Person named " +
           this.firstName + " " + this.lastName);
}
```

Here, we are actually combining 5 **String** objects to form a new **String** object containing the result ... the original 5 String objects remain unaltered.

Each character in a **String** is assigned an imaginary integer index that represents its order in the sequence. The first character in the **String** has an index of 0, the second character has

an index of 1, and so on. We can access any character from a **String** by using the **charAt()** method which requires us to specify the index of the character that we want to get:

```
String name = "Hank Urchif";
name.charAt(0);           // returns character 'H'
name.charAt(1);           // returns character 'a'
name.charAt(name.length() - 1); // returns character 'f'
name.charAt(name.length()); // causes StringIndexOutOfBoundsException
name.charAt(100);         // causes StringIndexOutOfBoundsException
```



There are also some methods in the **String** class that allow us to extract a sequence (or range) of characters from the String. The **substring(s,e)** method does just that. It takes two parameters **s** and **e**, where **s** specifies the starting character index and **e** specifies one more than the ending character index:

```
String name = "Hank Urchif";
name.substring(0, 4);           // returns character "Hank"
name.substring(5, 11);          // returns character "Urchif"
name.substring(1, name.length()); // returns character "ank Urchif"
name.substring(3, 6);           // returns character "k U"
```



In all cases above, the resulting **String** is a new object, the original **name** object remaining unchanged.

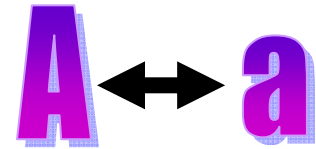
There is also a very useful method for eliminating unwanted leading and trailing characters (e.g., spaces, tabs, newlines and carriage returns). This can be useful when writing programs that get String input (e.g., name, address, email etc..) from the user through text fields on windows. The **trim()** method returns a new **String** object that represents the original string object but with no leading and trailing space, tab, newline or carriage return characters.

```
String s1 = "  I need a shave  ";
String s2 = " ";
s1.trim(); // returns "I need a shave"
s2.trim(); // returns empty string ""
```



Also, sometimes when getting input from the user we would like to force the information to be formatted as either uppercase or lowercase characters. Two useful methods called **toUpperCase()** and **toLowerCase()** will generate a copy of the string but with all alphabetic characters converted to uppercase or lowercase, respectively. The methods only affect the alphabetic characters ... all other characters remain the same.

```
String s = "Tea For 2!";
s.toUpperCase() ; // returns "TEA FOR 2!"
s.toLowerCase() ; // returns "tea for 2!"
```



A final important topic that we will discuss regarding strings is that of comparing strings with one another. String comparison is a fundamental tool used in many programs. For example, whenever we want to search for a person's name in a list, we must compare the name of the person (i.e., a **String**) with all of the names in a list of some sort.

JAVA has two useful methods for comparing Strings. The **equals(s)** method compares one **String** with another **String, s**, and then returns **true** if the two strings have the exact same characters in them and **false** otherwise. A similar comparison method called **equalsIgnoreCase(s)** is used to compare the two strings but in a way such that lowercase and uppercase characters are considered equal.

```
String apple1 = "apple";
String apple2 = "APPLE";
String apple3 = "apples";
String orange = "orange";

apple1.equals(orange);           // returns false
apple1.equals(apple2);          // returns false
apple1.equals(apple3);          // returns false
apple1.equals(apple2.toLowerCase()); // returns true

apple1.equalsIgnoreCase(apple2); // returns true
```



In regards to sorting strings, the **compareTo(s)** method will compare one string with another (i.e., parameter **s**) and return information about their respective alphabetical ordering. The method returns an integer which is:

- negative if the first string is *alphabetically before s*
- positive if the first string is *alphabetically after s*, or
- zero if the first string equals **s**

```
String apple = "Apple";
String orange = "Orange";
String banana = "Banana";

banana.compareTo(orange); // returns -13, Banana comes before Orange
banana.compareTo(apple); // returns 1, Banana comes after Apple
apple.compareTo("Apple"); // returns 0, Apple equals Apple
"Zebra".compareTo("apple"); // returns -7, uppercase chars are before lower!
"apple".compareTo("Apple"); // returns 32, lowercase chars are after upper!
```



You may notice, in the last two cases, that uppercase characters always come alphabetically before lowercase characters. You should always take this into account when sorting data.

To avoid sorting problems, it may be best to use `toUpperCase()` on each `String` before comparing them:

```
if (s1.toUpperCase().compareTo(s2.toUpperCase()) < 0)
    // s1 comes first
else
    // s2 comes first
```

Another very useful method in the `String` class is the `split()` method because it allows you to break up a `String` into individual substrings (called *tokens*) based on some separation criteria. For example, we can extract

- words from a sentence, one by one
- fields from a database or text file, separated by commas or other chars

The term *delimiter* is used to indicate the character(s) that separate the tokens (i.e., individual words or data elements).

Consider for example, the following `String` data which has been read in from a file:

```
"Mark,Lanthier,43,M,false"
```

Perhaps this is data for a particular person and we want to extract the information from the string in a way that we can use it. If we consider the comma ',' character as the only delimiter, then we can use the `split` method to obtain an array of `Strings` which we can then parse one by one to extract the needed data:

```
String s1 = "Mark,Lanthier,43,M,false";

String[] tokens = s1.split(",");
for (String token: tokens)
    System.out.println(token);
```



The code above will produce the following output:

```
Mark
Lanthier
41
M
false
```

Each token is an individual `String` that can be used afterwards. If, for example, we wanted to have just the 3rd piece of data (i.e., 41) and use it in a math expression, we could split the string and access just that piece of data, converting it to an integer as necessary ...

```
String    s1 = "Mark,Lanthier,43,M,false";
String[]  tokens;
int     age;

tokens = s1.split(",");
age = Integer.parseInt(tokens[2]);
if (age > 21) ...
```

The `","` parameter to the `split()` method above indicates that the `,` character is the delimiter. If we had the following String, however, we may want to include the `:` character as a delimiter as well:

```
"Mark,Lanthier:43:M,false"
```

We cannot simply use the parameter string `","` because that will only consider consecutive comma colon characters as delimiters (i.e., a 2-char delimiter). We want to allow the comma OR the colon to be delimiters, but not necessarily together. To accomplish this, the expression in the string becomes more complex. We basically have to indicate that we want all non-alphanumeric characters to be part of the tokens and everything else to be delimiters. So the following code would do what we want:

```
String    s1 = "Mark,Lanthier:43:'M',false";

String[]  tokens = s1.split("[^a-zA-Z0-9]");
for(String token: tokens)
    System.out.println(token);
```

Notice the square brackets `[]` in the parameter string. This indicates that we are about to list a sequence of characters to be the delimiters. The `^` character negates the list of characters to indicate that we are about to list all the non-delimiter characters (i.e., the token characters). Then we list the alphanumeric ranges `a-z`, `A-Z` and `0-9` to indicate that any alphanumeric character is part of a token, while everything else is to be considered a delimiter.

The parameter string is considered to be a *regular expression* (not discussed here) and can be quite complex. You may look in JAVA's API for more information. In some cases, the token strings will be of size 0. For example, consider the following code:

```
String    s1 = "Mark, Lanthier , 43 ,,, M , false";

String[]  tokens = s1.split("[, ]");    // comma or space delimiter
for(String token: tokens)
    System.out.println(token);
```

The following output would be obtained ...

```
Mark  
  
Lanthier  
  
41  
  
M  
  
false
```

Notice that there are many spaces in between. These spaces are empty strings. We should check for the empty strings in our code:

```
String s1 = "Mark, Lanthier , 43 ,,, M , false";  
  
String[] tokens = s1.split("[, ]"); // comma or space delimiter  
for(String token: tokens)  
    if (token.length() > 0)  
        System.out.println(token);
```

Then we obtain the output as before:

```
Mark  
Lanthier  
41  
M  
false
```

Supplemental Information (StringTokenizers)

There is another (perhaps simpler) way of extracting tokens from a **String** through use of the **StringTokenizer** class (imported from the **java.util** package). However, for some reason, the JAVA guys “suggest” that you use the **split()** method instead.

```
String s = "Mark, Lanthier , 44 ,,, M , false";

StringTokenizer tokens = new StringTokenizer(s, ", ");
System.out.println("The string has " + tokens.countTokens() + " tokens");

while(tokens.hasMoreTokens()) {
    System.out.println(tokens.nextToken());
}
```

This code will produce the same result as above, but with an extra line of output indicating the number of tokens in total, which is 5 in this example.

Interestingly, the **Scanner** class that we used for getting keyboard input can also be used to get tokens from a **String**. The list of delimiters however is actually a pattern sequence, not a list of separate delimiter characters. That means, whatever is listed as the delimiter string must match exactly (i.e., in the example below, a single comma must be followed by a single space character):

```
String sentence = "Banks, Rob, 34, Ottawa, 12.67";
Scanner s = new Scanner(sentence).useDelimiter(", ");
System.out.println(s.next());
System.out.println(s.next());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.nextFloat());
s.close();
```

Notice that the **Scanner** should be closed, we did not do this earlier but it is common practice.

13.2 The StringBuilder & Character Classes

Strings cannot be changed once created. Instead, when we try to manipulate them, we always get back a "brand new" **String** object. This is not normally a problem in most cases when programming, however, sometimes we would like to be able to *modify* a **String** by inserting/removing characters. For example, when we open a file in a text editor or word processor, we usually append, cut and insert text “on the fly”. It would be memory-inefficient and time-inefficient to continually create new strings and copy over characters from an old string to a new one.



The **StringBuilder** class in JAVA is useful for such a purpose. You may think of it simply as a **String** that can be modified. The **StringBuilder** methods run a little slower than their **String** equivalent methods, so if you plan to create strings that will not need to change, use **String** objects instead.

Here are two constructors for the **StringBuilder** class:

```
new StringBuilder();  
new StringBuilder(s); // s is a String
```

The first creates a **StringBuilder** with no characters to begin with and the second creates one with the characters equal to the ones in the given **String s**.

As with **Strings**, the **length()** method can be used to return the number of characters in the **StringBuilder** as follows:

```
StringBuilder sb1, sb2;  
  
sb1 = new StringBuilder();  
sb2 = new StringBuilder("hello there");  
sb1.length(); // returns 0  
sb2.length(); // returns 11
```

Unlike **Strings**, you can actually modify the length of the **StringBuilder** to any particular length by using a **setLength(int newLength)** method. If the **newLength** is less than the current length, the characters at the end of the **StringBuilder** are truncated. If the size is greater, **null** characters are used to fill in the extra places at the end as follows:

```
StringBuilder sb;  
  
sb = new StringBuilder("hello there");  
sb.setLength(9);  
System.out.println(sb); // displays "hello the"
```

As with **Strings**, the **charAt(int index)** method is used to access particular characters based on their **index** position (which starts at position 0). Unlike **Strings** though, a **setCharAt(int index, char c)** method is also available which allows you to change the character at the given **index** to become the specified character **c**. Here is how these methods work ...

```
StringBuilder name;  
  
name = new StringBuilder("Chip Electronic");  
name.charAt(3); // returns 'p'  
name.setCharAt(4, '+');  
System.out.println(name); // displays "Chip+Electronic"
```

However, a more commonly used method in the **StringBuilder** class is the **append(Object x)** method which allows you to append a bunch of characters to the end of the **StringBuilder**. If **x** is a **String** object, the entire string is appended to the end. If **x** is any other object, JAVA will call the **toString()** method for that object and append the resulting **String** to the end of the **StringBuilder**:

```
StringBuilder sb = new StringBuilder();
sb.append("Mark has ");
sb.append(new BankAccount("Mark"));
System.out.println(sb); // displays "Mark has Account #10000 with $0.0"
```

The resulting output may differ, of course, depending on the **BankAccount's toString()** method. Similar methods also exist for appending an **int**, **long**, **float**, **double**, **boolean** or **char** as follows:

```
append(int x), append(long x), append(float x),
append(double x), append(boolean x), append(char x)
```

The final two methods that we will mention allow you to remove characters from the **StringBuilder**. The **deleteCharAt(int index)** method will remove the character at the given **index** while the **delete(int start, int end)** method will delete all the characters within the indices ranging from **start** to **end-1** as follows:

```
StringBuilder sb;

sb = new StringBuilder("Miles Perlyter");
sb.delete(3,11); // changes sb to "Milter"
sb.deleteCharAt(sb.length()-1); // changes sb to "Milte"
sb.deleteCharAt(sb.length()-1); // changes sb to "Milt"
```

Sometimes, it is useful to use a **StringBuilder** to go through a **String** and make changes to it. For example, consider using a **StringBuilder** to remove all the non-alphabetic characters from a **String as follows** (of course the result would have to be a new **String**, since the original cannot be modified) ...

```
String original, result = "";
StringBuilder sb;
Character c;

original = "Hello, my 1st name ... is Mark !!";
sb = new StringBuilder();
for (int i=0; i<original.length(); i++) {
    c = original.charAt(i);
    if (Character.isLetter(c))
        sb.append(c);
}
result = new String(sb);
System.out.println(result);
```

Notice a couple of things from this code. First, the **StringBuilder** is used as a temporary object for creating the result string but is no longer useful after the method has completed. We use one of the **String** class' constructors to create the new **String** ... passing in the **StringBuilder**. Second, we are checking for non-alphabetic characters by using **Character.isLetter()**. Here, **isLetter()** is a **static** function in the **Character** class that determines whether or not the given character is alphabetic or not.



Side note: **Character** is a class in JAVA known as a **wrapper class** because it is an **object wrapper** for the **char** primitive. Essentially, the class can be used to “convert” (i.e., wrap up) a **char** into an object that can then be used as a regular object. There is a wrapper class for each of the primitives in JAVA (i.e., **Integer**, **Long**, **Float**, **Double**, **Character**, **Boolean**, **Short** and **Byte**). Since JAVA 1.5, primitives are automatically wrapped into objects, and so we need not worry about this.

There are other useful methods in the **Character** class. Here are just a few:

```
Character.isLetter(c)           // checks if c is a letter in the alphabet
Character.isDigit(c)           // checks if c is a digit (i.e., '0' - '9')
Character.isLetterOrDigit(c)   // ... this one is obvious ...
Character.isWhiteSpace(c)      // checks if c is the space character
Character.isLowerCase(c)       // checks if c is lowercase (e.g., 'a')
Character.isUpperCase(c)       // checks if c is uppercase (e.g., 'A')
Character.toLowerCase(c)       // returns lowercase equivalent of c
Character.toUpperCase(c)       // returns uppercase equivalent of C
```

Here are some examples of how they are used:

```
Character.isLetter('A')        // returns true
Character.isDigit('6')         // returns true
Character.isLetterOrDigit('@') // returns false
Character.isWhiteSpace(' ')    // returns true
Character.isLowerCase('a')     // returns true
Character.isUpperCase('A')     // returns true
Character.toLowerCase('B')     // returns 'b'
Character.toUpperCase('b')     // returns 'B'
```

Note that none of these methods require you to make an instance of a **Character** object. They are all **static**/class methods that take a **char** as a parameter (**int** in some cases) and return another primitive.

13.3 The Date and Calendar Classes

It is often necessary to use dates and times when programming. Let us take a look at the **Date** class provided in the **java.util** package. The **Date** class allows us to make data objects that incorporate time as well. The **java.util.Date** class is used to represent BOTH date and time. Dates are stored simply as a number, which happens to be the number of milliseconds since January 1, 1970, 00:00:00 GMT.



New dates are created with a call to a constructor as follows:

```
Date    today = new Date();
```

The result is an object that represents the current date and time and it looks something like this when displayed (of course it will vary depending on the day you run your code):

```
Thu Mar 26 14:39:17 EDT 2009
```

Notice that it shows the **day**, **month**, **day-of-month**, **hours**, **minutes**, **seconds**, **timezone** and **year** of the **Date** object. This is default behavior for this class. There are only three other useful methods in the **Date** class:

- `getTime()` - Returns a **long** representing this time in milliseconds.
- `after(Date d)` - Returns whether or not receiver date comes after the given date **d**.
- `before(Date d)` - Returns whether or not receiver date comes before the given date **d**.



Most other methods have been **deprecated** (which means they should not be used anymore).

In the class **Date** itself, there is no easy way to create a specific date (e.g., Feb. 13, 1992). Instead, we must use a different class to do this. In the current version of JAVA, **Calendar** objects are used to represent dates, instead of **Date** objects. **Calendar** is an abstract base class for converting between a **Date** object and a set of integer fields such as YEAR, MONTH, DAY, HOUR, and so on.



Although this **Calendar** class has many useful constants and methods (as you will soon see), we cannot make instances of it (i.e., we cannot say `new Calendar()`). Instead, the more specific kind of calendar called a **GregorianCalendar** is used.

The `java.util.GregorianCalendar` class is used to query and manipulate dates. Here are some of the available constructors ...

```
new GregorianCalendar()           // today's date
new GregorianCalendar(1999, 11, 31) // year, month, day
new GregorianCalendar(1968, 0, 8, 11, 55) // year, month, day, hours, mins
```

Notice that:

- the year is specified as 4-digits (e.g., 1968)
- months are specified from 0 to 11 (January being 0)
- days must be from 1 to 31
- hours and minutes are at the end of the constructor

Calendars do not display well.

Here is what you would see if you tried displaying a **GregorianCalendar**:

```
java.util.GregorianCalendar[time=1178909251343,areFieldsSet=true,
areAllFieldsSet=true, lenient=true, zone=sun.util.calendar.ZoneInfo[id=
"America/New_York",offset=-18000000,dstSavings=3600000,useDaylight=true,
transitions=235,lastRule=java.util.SimpleTimeZone[id=America/New_York,
offset=-18000000,dstSavings=3600000,useDaylight=true,startYear=0,
startMode=3,startMonth=3,startDay=1,startDayOfWeek=1,startTime=7200000,
startTimeMode=0,endMode=2,endMonth=9,endDay=-1,endDayOfWeek=1,endTime=
7200000,endTimeMode=0]],firstDayOfWeek=1,minimalDaysInFirstWeek=1,ERA=1,
YEAR=2007,MONTH=4,WEEK_OF_YEAR=19,WEEK_OF_MONTH=2,DAY_OF_MONTH=11,
DAY_OF_YEAR=131,DAY_OF_WEEK=6,DAY_OF_WEEK_IN_MONTH=2,AM_PM=1,HOURL=2,
HOURL_OF_DAY=14,MINUTE=47,SECOND=31,MILLISECOND=343,ZONE_OFFSET=
-18000000,DST_OFFSET=3600000]
```



Obviously, this is not pleasant. To display a **Calendar** in a friendlier manner, we can use the **getTime()** method, which actually returns a **Date** object (... not very intuitive ... I know). Consider these examples:

```
System.out.println(new GregorianCalendar().getTime()); // today
System.out.println(new GregorianCalendar(1999,11,31).getTime());
System.out.println(new GregorianCalendar(1968,0,8,11,55).getTime());
```

Here is the output (which of course varies with the current date):

```
Thu Mar 26 14:48:40 EDT 2009
Fri Dec 31 00:00:00 EST 1999
Mon Jan 08 11:55:00 EST 1968
```



The **isLeapYear(int year)** method returns whether or not the given year is a leap year for this calendar:

```
new GregorianCalendar().isLeapYear(2008); // returns true
new GregorianCalendar().isLeapYear(2009); // returns false
```



There are many other methods that we can use to query or alter the date which are inherited from the **Calendar** class.

For example, the **get(int field)** method is used along with some **static** constants to access information about the particular calendar date. For example, at the time of updating these notes the date was:

```
Thu Mar 26 15:05:35 EDT 2009
```

Consider the results (shown to the right) of each **get** method call in the code below. You should use **import java.util.Calendar** at the top of your code so that you can use these constants:

```

Calendar today = Calendar.getInstance();

today.get(Calendar.YEAR);           // 2009
today.get(Calendar.MONTH);         // 2
today.get(Calendar.DAY_OF_MONTH);  // 26
today.get(Calendar.DAY_OF_WEEK);   // 5
today.get(Calendar.DAY_OF_WEEK_IN_MONTH); // 4
today.get(Calendar.DAY_OF_YEAR);   // 85
today.get(Calendar.WEEK_OF_MONTH); // 4
today.get(Calendar.WEEK_OF_YEAR);  // 13
today.get(Calendar.HOUR);          // 3
today.get(Calendar.AM_PM);         // 1
today.get(Calendar.HOUR_OF_DAY);   // 15
today.get(Calendar.MINUTE);        // 5
today.get(Calendar.SECOND);        // 35

```


The value returned from the **get(int field)** method can be compared with other **Calendar** constants. For example,

```

if (aCalendar.get(Calendar.MONTH) == Calendar.APRIL) {...}
if (aCalendar.get(Calendar.DAY_OF_WEEK) == Calendar.SATURDAY) {...}

```

Here are some of the useful constants:

Calendar.SUNDAY	Calendar.JANUARY	Calendar.JULY	
Calendar.MONDAY	Calendar.FEBRUARY	Calendar.AUGUST	
Calendar.TUESDAY	Calendar.MARCH	Calendar.SEPTEMBER	
Calendar.WEDNESDAY	Calendar.APRIL	Calendar.OCTOBER	
Calendar.THURSDAY	Calendar.MAY	Calendar.NOVEMBER	
Calendar.FRIDAY	Calendar.JUNE	Calendar.DECEMBER	
Calendar.SATURDAY	Calendar.AM	Calendar.PM	

There is also a **set(int field, int value)** method that can be used to set the values for certain date fields:

```

aCalendar.set(Calendar.MONTH, Calendar.JANUARY);
aCalendar.set(Calendar.YEAR, 1999);
aCalendar.set(Calendar.AM_PM, Calendar.AM);

```

Other set methods allow the date and time to be changed ...

```

aCalendar.set(1999, Calendar.AUGUST, 15);
aCalendar.set(1999, Calendar.AUGUST, 15, 6, 45);

```



We can also format dates when we want to print them nicely. There is a **SimpleDateFormat** class (in the **java.text** package) that formats a **Date** object using one of many predefined formats. It does this by generating a **String** representation of the date. The constructor takes a **String** which indicates the desired format:

```

new SimpleDateFormat("MMM dd, yyyy");

```

The parameter in the method is a format string that specifies “how you want the date to look” when it is printed. By using different characters in the format string, you get different output for the date. The `format(Date d)` method in the `SimpleDateFormat` class is then used to actually do the work by applying the format to the given date. Here is an example:

```
import java.text.SimpleDateFormat;
...
SimpleDateFormat dateFormatter = new SimpleDateFormat("MMM dd, yyyy");
Date today = new Date();
String result = dateFormatter.format(today);

System.out.println(result);
```

Here is the result (which would vary, depending on the date):

```
Mar 26, 2009
```

Here are examples of format Strings and their effect on the date April 30th 2001 at 12:08 PM:

Format String	Resulting output
without formatting	Tue Apr 10 15:07:52 EDT 2001
"yyyy/MM/dd"	2001/04/30
"yy/MM/dd"	01/04/30
"MM/dd"	04/30
"MMM dd, yyyy"	Apr 30, 2001
"MMMM dd, yyyy"	April 30, 2001
"EEE. MMMM dd, yyyy"	Mon. April 30, 2001
"EEEE, MMMM dd, yyyy"	Monday, April 30, 2001
"h:mm a"	12:08 PM
"MMMM dd, yyyy (hh:mma)"	April 30, 2001 (12:08PM)

For additional formatting information, check out the JAVA API specification. Here is a simple example that creates two dates. One representing today, the other representing a future date:

```
import java.util.*;
import java.text.SimpleDateFormat;

public class DateTestProgram {
    public static void main (String[] args) {

        Calendar today = Calendar.getInstance();
        Calendar future;
        int difference;

        // Display Information about today's date and time
        System.out.println("Here is today:");
        System.out.println(today.getTime());
```

```

System.out.println(today.get(Calendar.YEAR));
System.out.println(today.get(Calendar.MONTH));
System.out.println(today.get(Calendar.DAY_OF_MONTH));

// Display Information about a future day's date and time
future = Calendar.getInstance();
future.set(2010, Calendar.MARCH, 5);
System.out.println("Here is the future:");
System.out.println(future.getTime());
System.out.println(future.get(Calendar.YEAR));
System.out.println(future.get(Calendar.MONTH));
System.out.println(future.get(Calendar.DAY_OF_MONTH));

// Test the formatting
Date aDate = new Date();
System.out.println(aDate);
System.out.println(new SimpleDateFormat("yyyy/MM/dd").format(aDate));
System.out.println(new SimpleDateFormat("yy/MM/dd").format(aDate));
System.out.println(new SimpleDateFormat("MM/dd").format(aDate));
System.out.println(new SimpleDateFormat("MMM dd, yyyy").format(aDate));
System.out.println(new SimpleDateFormat("MMMM dd, yyyy").format(aDate));
}
}

```

Here is the output from running this code on May 30th, 2011:

```

Here is today:
Mon May 30 15:37:47 EDT 2011
2011
4
30
Here is the future:
Fri Mar 05 15:37:47 EST 2010
2010
2
5
Mon May 30 15:37:47 EDT 2011
2011/05/30
11/05/30
05/30
May 30,2011
May 30,2011

```

Notice that the months start at **0**, and so March is month **#2**.

Although we can create and display simple dates, we have not done any manipulation at all. For instance, we may want to know how many working days there are between two dates. There are many more functions in the **Calendar** and **Date** classes, but we will not discuss them any further here. You would have to look at the API for the **Date**, **Calendar**, **GregorianCalendar** and **SimpleDateFormat** classes.

Supplemental Information (Formatting Dates with Strings)

We can also use the **String.format()** method to format dates and times. There are many flags that can be used (see the API for details) but here are some commonly used ones for displaying dates and times:

```
Date aDate = new Date();

System.out.println(String.format("%tc", aDate));
System.out.println(String.format("%tF", aDate));
System.out.println(String.format("%tR", aDate));
System.out.println(String.format("%tr", aDate));
System.out.println(String.format("%tD", aDate));
```

Here was the output when it was ran on March 26, 2009 at 3:26pm:

```
Thu Mar 26 15:26:56 EDT 2009
2009-03-26
15:26
03:26:56 PM
03/26/09
```

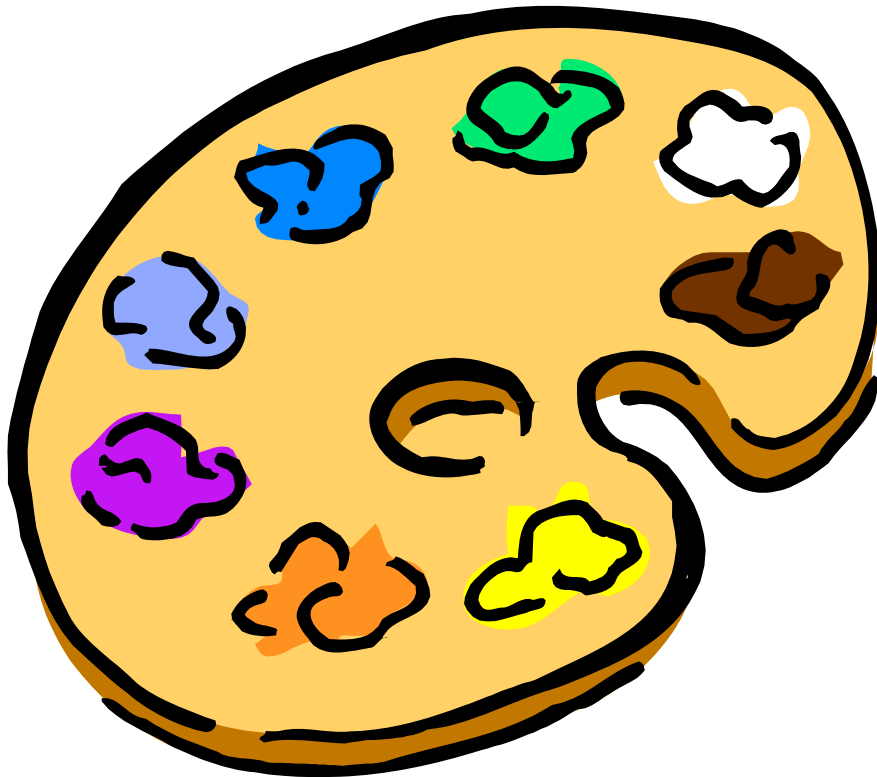
This page has been intentionally left blank.

Chapter 14

Graphics

What is in This Chapter ?

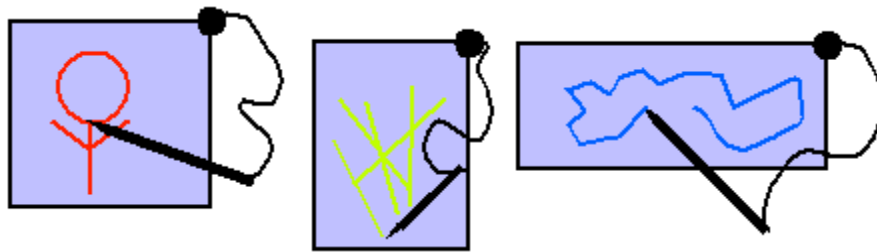
As programmers, we will likely all eventually come across a situation in which we need to display graphics. Graphics may be pictures or perhaps drawings consisting of lines, circles, rectangles etc... For example, if we want to have an application that displays a bar graph, there is no "magical" component in JAVA that does this for us. In this chapter, we will learn the basics of **drawing graphics**, **displaying images**, and **manipulating graphics** in our JAVA applications.



14.1 Doing Simple Graphics

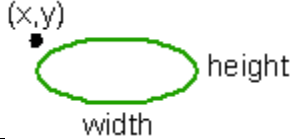


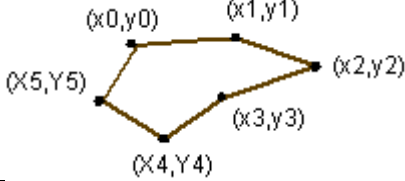
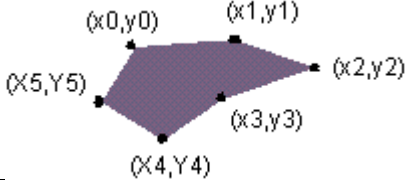


Graphics are used in many applications to display graphs, statistics, diagrams, pictures etc... Some applications are even completely based on graphics such as games, paint programs, MS PowerPoint etc... We have already seen that **ImageIcons** can be used to display images on your application window inside labels, buttons etc... Now we will see how to actually draw our own graphics, as when drawing graphs or diagrams.

The **java.awt** package has a class called **Graphics** that permits the drawing of various shapes. The class is **abstract** and so there is no constructor. Instead, JAVA provides a **getGraphics()** method that can be sent to any window component which returns an instance of this **Graphics** class (i.e., each component keeps an instance of that class by default). Think of each component having its own "pen" that can only be used to draw in that component's "space", just like the pens attached to kiosks at the bank.



There are a set of drawing functions that allow you to draw onto a component's area. Since a particular graphics object belongs to one specific component, you can only draw on that component with it. Most drawing functions allow you to specify x and y coordinates. The coordinate (x,y)=(0,0) is at the top left corner of the component's area. So all coordinates are with respect to the component's area. Here are just some of the methods available in the **Graphics** class (look in the JDK API for more info):

	<pre>// Draw a line from (x1, y1) to (x2,y2) public abstract void drawLine(int x1, int y1, int x2, int y2);</pre>
	<pre>// Draw a rectangle with its top left at (x, y) having the given width and height public abstract void drawRect(int x, int y, int width, int height);</pre>
	<pre>// Draw a filled rectangle with its top left at (x, y) having the given width and height public abstract void fillRect(int x, int y, int width, int height);</pre>
	<pre>// Erase a rectangular area by filling it in with the background color public abstract void clearRect(int x, int y, int width, int height);</pre>

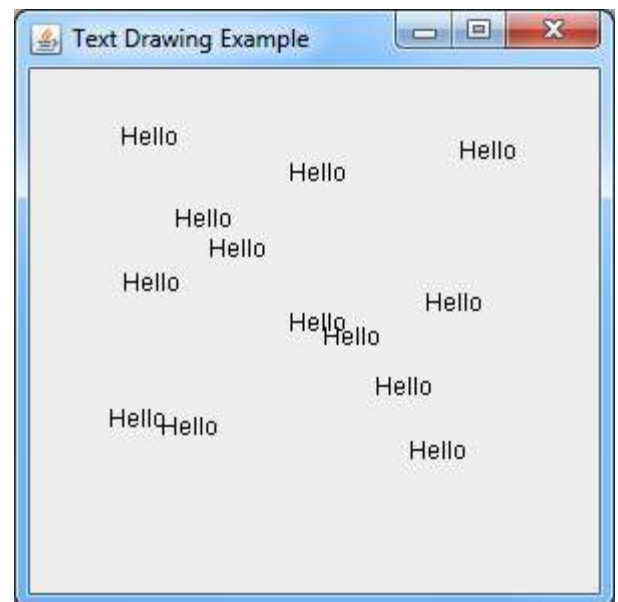
	<pre>// Draw an oval with its top left at (x, y) having the given width and height public abstract void drawOval(int x, int y, int width, int height);</pre>
	<pre>// Draw a filled oval with its top left at (x, y) having the given width and height public abstract void fillOval(int x, int y, int width, int height);</pre>
	<pre>// Draw the given String with its bottom left at (x, y) public abstract void drawString(String str, int x, int y);</pre>
	<pre>// Draw a polygon with the given coordinates public abstract void drawPolygon(int[] x, int[] y, int numEdges);</pre>
	<pre>// Draw a filled polygon with the given coordinates public abstract void fillPolygon(int[] x, int[] y, int numEdges);</pre>
	<pre>// Set the foreground & fill color of Graphics object public abstract void setColor(Color c);</pre>
	<pre>// Set the Font for use with drawString public abstract void setFont(Font font);</pre>

Example:

This code makes a simple **JFrame** and then draws some text on it wherever the user clicks the mouse. As it turns out, we can draw directly to the frame of a window. We don't need to add any components for this example. To the right is a snapshot of the running program.

You will notice three things about this example:

1. The text is drawn such that the bottom left corner of the text appears at the location which the mouse is clicked.
2. The text is erased whenever we alter the size of the window.
3. We can ask a **MouseEvent** for the **x** and **y** position of the mouse.



Here is the code:

```

import java.awt.event.*;
import javax.swing.*;

public class TextDrawingExample extends JFrame {

    public TextDrawingExample(String title) {
        super(title);
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                getGraphics().drawString("Hello", e.getX(), e.getY());
            }
        });

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300, 300);
    }

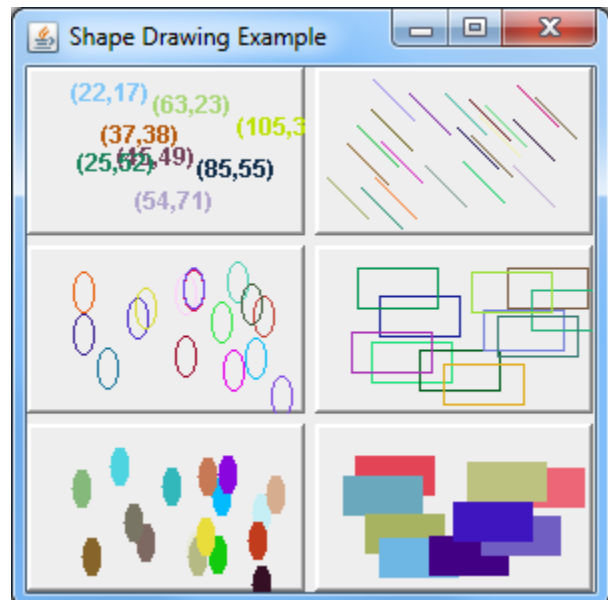
    public static void main(String args[]) {
        new TextDrawingExample("Text Drawing Example").setVisible(true);
    }
}

```

Example:

In this example, we will set up six **JLabels**, each one allowing a different shape to be drawn onto it. We will set up a single event handler for all mouse presses and within that method we will ask which label has been clicked on and then draw the corresponding shape onto the label. The shapes will be drawn with different colors each time. We use **Math.random()** to get a random number for creating a random color. To the right is a snapshot of the working program. You will notice that:

1. The **getGraphics()** message is sent to the component, not to the frame.
2. The labels have neat little borders which were created by using



BorderFactory.createRaisedBevelBorder().

You can take a look at the Java API to find out more about the different kinds of borders that are possible.

Here is the code:

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

```

```

public class ShapeDrawingExample extends JFrame {
    private JLabel labels[];

    public ShapeDrawingExample(String title) {
        super(title);
        setLayout(new GridLayout(3,2,5,5));
        labels = new JLabel[6];
        for (int i=0; i<6; i++) {
            getContentPane().add(labels[i] = new JLabel());
            labels[i].setBorder(BorderFactory.createRaisedBevelBorder());
        }
        addListeners();
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300, 300);
    }

    // Add listener for a mouse press
    private void addListeners() {
        MouseAdapter anAdapter = new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                JLabel area = (JLabel)e.getSource();
                Graphics g = area.getGraphics();

                // Get a random color
                g.setColor(new Color((float)Math.random(),
                                     (float)Math.random(),
                                     (float)Math.random()));

                // Find the label that caused this event
                int labelNumber;
                for (labelNumber=0; labelNumber<6; labelNumber++) {
                    if (area == labels[labelNumber]) break;
                }
                int x = e.getX();
                int y = e.getY();

                // Now decide what to draw
                switch (labelNumber) {
                    case 0: g.drawString("(" + String.valueOf(x) + "," +
                                         String.valueOf(y) + ")", x, y); break;
                    case 1: g.drawLine(x, y, x+20, y+20); break;
                    case 2: g.drawOval(x, y, 10, 20); break;
                    case 3: g.drawRect(x, y, 40, 20); break;
                    case 4: g.fillOval(x, y, 10, 20); break;
                    case 5: g.fillRect(x, y, 40, 20); break;
                }
            }
        };

        // Add mouse listeners to all labels
        for (int i=0; i<6; i++)
            labels[i].addMouseListener(anAdapter);
    }

    public static void main(String args[]) {
        new ShapeDrawingExample("Shape Drawing Example").setVisible(true);
    }
}

```

14.2 Repainting Components

You may have noticed in our examples so far that all the drawings we do are erased when the window is resized. When a window is resized, each of the components needs to be redrawn. Every **JComponent** has (or inherits) a **repaint()** method which is called by JAVA automatically when the window is resized in order to redraw the component. JAVA redraws these components as it already knows how to do, but it will not automatically redraw anything that we may have drawn manually, unless we tell it to. In fact, we too can call this **repaint()** method any time we want our component to be redrawn.

The **repaint()** method actually calls a method called **paintComponent(Graphics g)**, which is also inherited from the **JComponent** class. However, the default inherited **paintComponent()** method does not know what you want to be painted. In order to tell it what to actually redraw, you need to override this method by writing your own **paintComponent()** method which will specify exactly how to draw your graphics.

To add this functionality to our previous two examples, we would have to "keep track of" all the graphical shapes that we have been drawing (as well as their attributes, such as location, dimension and color) so that in our **paintComponent()** method, we can redraw all of them properly each time.

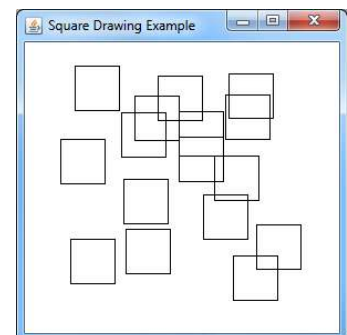
The previous two examples showed how simple graphics can be drawn effortlessly on a frame or on a label. In fact, you can draw on any component. The component that is intended for general purpose drawing is a **JPanel**.

Note in the older AWT framework of JAVA, a special class called a **Canvas** was used for drawing using a **paint()** method, not the **paintComponent()** method. **JPanels** in the newer Swing library have all the capabilities of the old **Canvas** class built-in and should be used instead. In fact if the older **paint()** method is used you can expect bugs, so use the **JPanels** and **paintComponent()** method instead.

The common strategy in JAVA for drawing on a blank area is to make your own class which is a subclass of **JPanel**. This class should implement, or override, the **paintComponent()** method. When we override this method however, we will be sure to call the **super** method so that the default drawing of the component still occurs.

Example:

In this example, we create a subclass of **JPanel** on which we will keep track of mouse click locations and draw 40x40 pixel squares centered at each of these locations. We will override the **paintComponent()** method so that the squares will be properly redrawn whenever we (or JAVA) call **repaint()** or when the window is resized. The application itself is not so exciting to look at, but rather the underlying concept of painting on the panel is what is important.



In this example, you may notice a couple of things:

- The `getPoint()` method is sent to a **MouseEvent** object to obtain the **Point** object representing the location that was clicked.
- Since all squares will be the same size, we don't store the size, just their center locations.

Here is the code:

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class SquareCanvas extends JPanel implements MouseListener {

    private ArrayList<Point> squares;    // Keep track of all square centers

    public SquareCanvas() {
        squares = new ArrayList<Point>();
        setBackground(Color.white);
        addMouseListener(this);
    }

    // Displays the contents of the canvas
    public void paintComponent(Graphics graphics) {
        // Draw the component as before (i.e., default look)
        super.paintComponent(graphics);
        // Now draw all of our squares
        graphics.setColor(Color.black);
        for (Point center: squares)
            graphics.drawRect(center.x - 20, center.y - 20, 40, 40);
    }

    // These are unused MouseEventHandlers. Note that we could have
    // used an Adapter class here. However, a typical drawing
    // application would make use of these other events as well.
    public void mouseClicked(MouseEvent event) {}
    public void mouseEntered(MouseEvent event) {}
    public void mouseExited(MouseEvent event) {}
    public void mouseReleased(MouseEvent event) {}

    // Store the mouse location when it is pressed
    public void mousePressed(MouseEvent event) {
        squares.add(event.getPoint());
        repaint(); // this will call paintComponent()
    }

    public static void main(String args[]) {
        JFrame frame = new JFrame("Square Drawing Example");
        frame.add(new SquareCanvas());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 300);
        frame.setVisible(true);
    }
}
```

Notice how we are redrawing the panel by first making a call to **super.paintComponent()**. This ensures that the panel's background is redrawn (i.e., erased) before we start drawing again. If we did not do this call, our squares would still be drawn, but the background color for the window (i.e., white in this case) would not be shown. We would end up with the light gray default background coloring of the **JFrame**.

14.3 Displaying Images

We have seen how to draw shapes of different colors onto components, now we will find out how to draw an image on the screen. JAVA lets you load and display both ".gif" files as well as ".jpg" files. We have seen the use of **ImageIcons** with components so that we can display an icon along with text or as a label of a button. Icons, however, are meant to be small images and are not meant for large images. When larger pictures are to be shown, you should use **Image** objects. In fact, the **Image** class is abstract, but there are two useful subclasses. In JAVA, there is much to learn about **Image** objects. There are many classes relating to the manipulation of images and a thorough investigation into these classes is well beyond the scope of this course. Here, we will look simply at the basic displaying of images in our applications.

A typical scenario is to load and display an image (such as a .gif or .jpg) from a file. Unfortunately, the way images are obtained from files is a platform-specific issue. This means that it is not always done the same way, depending on what machine you run your code. Fortunately, JAVA supplies a **Toolkit** class that has common "special" methods for doing various platform-specific things such as loading images.

We can load an image from the disk by asking the **Toolkit** class for an instance of **Toolkit** (i.e., default will do fine) and then get the image as follows:

```
Image myImage = Toolkit.getDefaultToolkit().createImage("picture.gif");
```

The code loads and returns an **Image** object from the file entitled **picture.gif** but it does not display the image. We can display the image by asking a **Graphics** object to draw the image:

```
g.drawImage(anImage, x, y, null);
```

The image is drawn with its top-left corner at (x, y) in this graphics context's coordinate space. The 4th parameter can be any class that implements the **ImageObserver** interface. This interface is used as a means of informing a class when an image is done being loaded or drawn (since images in general may take a while to load or draw ... especially if being loaded from a network). This strategy of informing interested classes of image completion, allows more efficient use of process cycles so that the program does not sit idly by doing nothing while the image is being loaded/drawn. We will keep things simple in our example and set this value to **null** so that nobody is informed when the image is loaded or drawn.

One final issue that we are interested in is with respect to the image size. We may want to create a **JPanel** that has the exact same size of the image (e.g., for use as a background image for the panel). In this case, we can ask an image for its width and height before

choosing the size of our panel. There are **getWidth()** and **getHeight()** methods that we can send to our Image object to obtain these values. However, there is one minor issue. While the image is being loaded (which may take a while), the value returned from **getWidth()** and **getHeight()** is -1. So, we have to introduce a delay in our program by waiting until these methods return valid results:

```
while ((anImage.getWidth(null) == -1) && (anImage.getHeight(null) == -1));
```

Notice as well that these methods take an **ImageObserver** as a parameter (which we set to **null**). By using a proper **ImageObserver**, we would not have to put in this delay, but could perform other application-specific tasks while we wait for the image to be loaded.

Now we may set the "preferred size" of the panel. Note that setting the "size" of the panel is not useful since when placed on a frame, the frame's layout manager will automatically resize all of its components.

```
setPreferredSize(new Dimension(anImage.getWidth(null), anImage.getHeight(null)));
```

So here is the code we can use to test:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ImagePanel extends JPanel {
    private Image anImage;

    public ImagePanel() {
        anImage = Toolkit.getDefaultToolkit().createImage("altree.gif");

        while ((anImage.getWidth(null) == -1) && (anImage.getHeight(null) == -1));
        setPreferredSize(new Dimension(anImage.getWidth(null),
                                       anImage.getHeight(null)));
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawImage(anImage, 0, 0, null);
    }

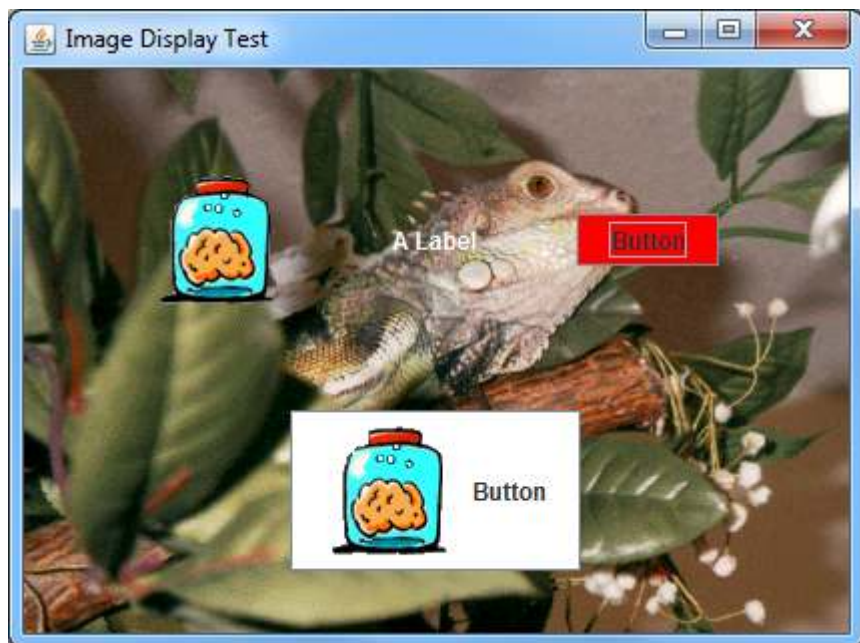
    public static void main(String args[]) {
        JFrame frame = new JFrame("Image Display Test");
        frame.add(new ImagePanel());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack(); // Makes size according to panel's preference
        frame.setVisible(true);
    }
}
```

Here is the result:

By the way, this is "AL".
She was my first Iguana.
She is no longer alive, but now
she is remembered here in our
notes :).



Note that since we used the panel's **paintComponent()** method to draw the image, the image is drawn as a background and so any components we add to the panel will appear on top of the image. So you can see that it is quite easy to create a window as shown below simply by adding components to the panel as usual:



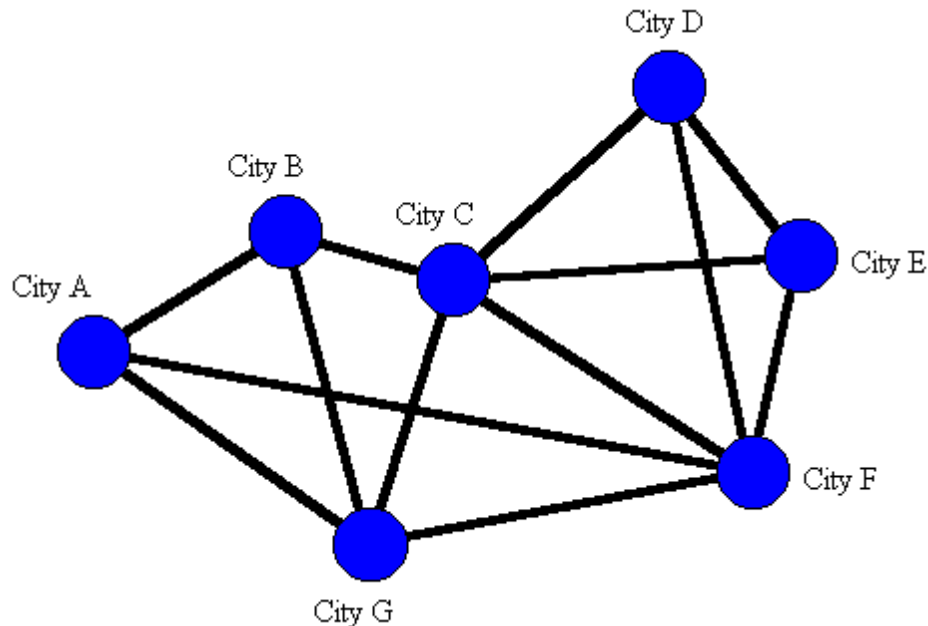
There are many more things that you can do with images:

- Shrink/Grow them
- Fade them
- Warp them
- other filters ...

We do not have time to fully investigate these other features of the API library. Feel free to experiment on your own.

14.4 Graph Editor Example

This section of the notes describes a step-by-step approach for creating a simple graph editor. It introduces the notion of "drag and drop" as well as selecting objects.



What is a graph ? There are many types of graphs. We are interested in graphs that form topological and/or spatial information. Our graphs will consist of **nodes** and **edges**. The nodes may represent cities in a map while the edges may represent roads between cities:

We would like to make a graph editor with the ability to:

- add/remove nodes
- add/remove edges
- move nodes around (edges between them will remain connected)
- "select" groups of nodes and edges for removal or moving
- do some other useful graph-manipulation features

The Graph Model:

We will begin our application as usual by developing the model. We know that our graph itself is going to be the model, but we must first think about what components make up the graph. These are the nodes and edges.

Let us begin by creating a **Node** class. What **state** should each node maintain ? Well, it depends on the application that will be using it. Since we know that the graph will be displayed, each node will need to keep track of its **location**. Also, we may wish to **label** nodes (e.g., a city's name). Here is a basic model for the nodes:

```

import java.awt.Point;

public class Node {
    private String    label;
    private Point     location;

    public Node() { this("", new Point(0,0)); }
    public Node(String aLabel) { this(aLabel, new Point(0,0)); }
    public Node(Point aPoint) { this("", aPoint); }
    public Node(String aLabel, Point aPoint) {
        label = aLabel;
        location = aPoint;
    }

    public String getLabel() { return label; }
    public Point getLocation() { return location; }
    public void setLabel(String newLabel) { label = newLabel; }
    public void setLocation(Point aPoint) { location = aPoint; }
    public void setLocation(int x, int y) { location = new Point(x, y); }

    // Nodes look like this: label(12,43)
    public String toString() {
        return(label + "(" + location.x + "," + location.y + ")");
    }
}

```

Notice that we don't have much in terms of behavior ... simply some get/set methods and a **toString()** method. Notice the two different set methods for location. This gives us flexibility in cases where we the coordinates are either **Point** objects or **ints**.

What state do we need for a graph edge? Well ... they must *start* at some node and *end* at another so we may want to know which nodes these are. Does it make sense for a graph edge to exist when one or both of its endpoints are not nodes? Probably not. So an edge should keep track of the node from which it starts and the node at which it ends. We will call them **startNode** and **endNode**. What about a label? Sure ... roads have names (as well as lengths). Here is a basic **Edge** class:

```

public class Edge {
    private String    label;
    private Node     startNode, endNode;

    public Edge(Node start, Node end) { this("", start, end); }
    public Edge(String aLabel, Node start, Node end) {
        label = aLabel;
        startNode = start;
        endNode = end;
    }

    public String getLabel() { return label; }
    public Node getStartNode() { return startNode; }
    public Node getEndNode() { return endNode; }
}

```

```

public void setLabel(String newLabel) { label = newLabel; }
public void setStartNode(Node aNode) { startNode = aNode; }
public void setEndNode(Node aNode) { endNode = aNode; }

// Edges look like this: sNode(12,43) --> eNode(67,34)
public String toString() {
    return(startNode.toString() + " --> " + endNode.toString());
}
}

```

Now what about the graph itself ? What do we need for the state of the graph ?
Well ... a graph is just a bunch of nodes and edges.

Still, we have a few choices for representing the Graph:

1. Keep a collection of all nodes AND another collection of all edges
2. Keep only a collection of all nodes
3. Keep only a collection of all edges
4. Keep only 1 node OR 1 edge (this seems weird doesn't it ?)

Let us examine each of these:

1. The 1st strategy would provide quick access for nodes and edges since they are readily available. However, it does take more space than the other strategies.
2. The 2nd strategy allows quick access to nodes, but if we ever needed to get all the edges, we would have to build up the collection, which takes time. This can be done by iterating through all *incident edges* of all nodes and adding the edges (this is slower, but more space efficient). So each node would have to keep track of the edges from/to it.
3. The 3rd strategy is similar to the 2nd except that the edges are efficiently accessible and the nodes are not.
4. The 4th strategy is weird. If we keep one node, we would have to traverse along one of its *incident edges* to the other end and continue in this manner throughout the graph in order to collect all the nodes or edges. However, this will ONLY work if the graph is **connected** (i.e., every node can be reached from every other node through a sequence of graph edges).

We will choose the 2nd strategy for our implementation, although you should realize that all three are possible.

Let us examine our **Node** and **Edge** classes a little further and try to imagine additional behavior that we may want to have.

Notice that each edge keeps track of the nodes that it connects to. But shouldn't a node also keep track of the edges are connected to it ? Think of "real life". Wouldn't it be nice to know which roads lead "into" and "out of" a city ?

Obviously, we can always consult the graph itself and check ALL edges to see if they connect to a given city. This is NOT what you would do if you had a map though. You don't find this

information out by looking at ALL roads on a map. You find the city of interest, then look at the roads around that area (i.e., only the ones heading into/out of the city).

The point is ... for time efficiency reasons, we will probably want each node to keep track of the edges that it is connected to. Of course, we won't make copies of these edges, we will just keep "pointers" to them so the additional memory usage is not too bad.

We should go back and add the following instance variable to the **Node** class:

```
private ArrayList<Edge>    incidentEdges;
```

We will also need the following "get method" and another for adding an edge:

```
public ArrayList<Edge> incidentEdges() {
    return incidentEdges;
}
```

```
public void addIncidentEdge(Edge e) {
    incidentEdges.add(e);
}
```

We will also have to add this line to the last of the **Node** constructors:

```
incidentEdges = new ArrayList<Edge>();
```

While we are making changes to the **Node** class, we will also add another interesting method called **neighbours** that returns the nodes that are connected to the receiver node by a graph edge. That is, it will return an **ArrayList** of all nodes that share an edge with this receiver node. It is very much like asking: "which cities can I reach from this one if I travel on only one highway?".

We can obtain these neighbors by iterating through the **incidentEdges** of the receiver and extracting the node at the other end of the edge. We will have to determine if this other node is the start or end node of the edge:

```
public ArrayList<Node> neighbours() {
    ArrayList<Node> result = new ArrayList<Node>();

    for (Edge e: incidentEdges) {
        if (e.getStartNode() == this)
            result.add(e.getEndNode());
        else
            result.add(e.getStartNode());
    }

    return result;
}
```

As we write this method, it seems that we are writing a portion of code that is *potentially* useful for other situations. That code is the code responsible for finding the opposite node of an edge. We should extract this code and make it a method for the **Edge** class:

```

public Node otherEndFrom(Node aNode) {
    if (startNode == aNode)
        return endNode;
    else
        return startNode;
}

```

Now, we can rewrite the `neighbours()` method to use the `otherEndFrom()` method:

```

public ArrayList<Node> neighbours() {
    ArrayList<Node> result = new ArrayList<Node>();

    for (Edge e: incidentEdges)
        result.add(e.otherEndFrom(this));

    return result;
}

```

Ok. Now we will look at the **Graph** class. We have decided that we were going to store just the nodes, and not the edges. We will also store a label for the graph ... after all ... provinces have names don't they ?

```

import java.util.*;

public class Graph {
    private String          label;
    private ArrayList<Node> nodes;

    public Graph() { this("", new ArrayList<Node>()); }
    public Graph(String aLabel) { this(aLabel, new ArrayList<Node>()); }
    public Graph(String aLabel, ArrayList<Node> initialNodes) {
        label = aLabel;
        nodes = initialNodes;
    }
    public ArrayList<Node> getNodes() { return nodes; }
    public String getLabel() { return label; }
    public void setLabel(String newLabel) { label = newLabel; }

    // Graphs look like this: label(6 nodes, 15 edges)
    public String toString() {
        return (label + "(" + nodes.size() + " nodes, " +
            getEdges().size() + " edges)");
    }
}

```

Let us write a method to return all the edges of the graph. It will have to go and collect all the **Edge** objects from the incident edges of the **Node** objects and return them as an **ArrayList**. Can you foresee a small problem ?

```
// Get all the edges of the graph by asking the nodes for them
public ArrayList<Edge> getEdges() {
    ArrayList<Edge> edges = new ArrayList<Edge>();

    for (Node n: nodes) {
        for (Edge e: n.incidentEdges()) {
            if (!edges.contains(e)) //so that it is not added twice
                edges.add(e);
        }
    }

    return edges;
}
```

Now we need methods for adding/removing nodes/edges. Adding a node or edge is easy, assuming that we already have the node or edge:

```
public void addNode(Node aNode) {
    nodes.add(aNode);
}
```

```
public void addEdge(Edge anEdge) {
    // ?????? What ?????? ...
}
```

Wait a minute ! How do we add an edge if we do not store them explicitly ? Perhaps we don't want an **addEdge** method that takes an "already created" edge. Instead, we should have an **addEdge** method that takes the **startNode** and **endNode** as parameters, then it creates the edge:

```
public void addEdge(Node start, Node end) {
    // First make the edge
    Edge anEdge = new Edge(start, end);

    // Now tell the nodes about the edge
    start.addIncidentEdge(anEdge);
    end.addIncidentEdge(anEdge);
}
```

There ... that is better. What about removing/deleting a node or edge ? Deleting an **Edge** is easy, we just ask the edge's start and end nodes to remove the edge from their lists. Removing a **Node** is a little more involved since all of the incident edges must be removed as well. After all ... we cannot have edges dangling with one of its **Nodes** missing !

```
public void deleteEdge(Edge anEdge) {
    // Just ask the nodes to remove it
    anEdge.getStartNode().incidentEdges().remove(anEdge);
    anEdge.getEndNode().incidentEdges().remove(anEdge);
}
```

```

public void deleteNode(Node aNode) {
    // Remove the opposite node's incident edges
    for (Edge e: aNode.incidentEdges())
        e.otherEndFrom(aNode).incidentEdges().remove(e);

    nodes.remove(aNode); // Remove the node now
}

```

OK. Let us write some code that now tests the model classes. Here is **static** method for the **Graph** class that creates and returns a graph:

```

public static Graph example() {
    Graph myMap = new Graph("Ontario and Quebec");
    Node ottawa, toronto, kingston, montreal;

    myMap.addNode(ottawa = new Node("Ottawa", new Point(250,100)));
    myMap.addNode(toronto = new Node("Toronto", new Point(100,170)));
    myMap.addNode(kingston = new Node("Kingston", new Point(180,110)));
    myMap.addNode(montreal = new Node("Montreal", new Point(300,90)));
    myMap.addEdge(ottawa, toronto);
    myMap.addEdge(ottawa, montreal);
    myMap.addEdge(ottawa, kingston);
    myMap.addEdge(kingston, toronto);

    return myMap;
}

```

We can test it by writing **Graph.example()** anywhere. This looks fine and peachy, but if we have 100 nodes, we would need 100 local variables (or a big array) just for the purpose of adding edges !! Maybe this would be a better way to write the code:

```

public static Graph example() {
    Graph myMap = new Graph("Ontario and Quebec");

    myMap.addNode(new Node("Ottawa", new Point(250,100)));
    myMap.addNode(new Node("Toronto", new Point(100,120)));
    myMap.addNode(new Node("Kingston", new Point(200,130)));
    myMap.addNode(new Node("Montreal", new Point(300,70)));
    myMap.addEdge("Ottawa", "Toronto");
    myMap.addEdge("Ottawa", "Montreal");
    myMap.addEdge("Ottawa", "Kingston");
    myMap.addEdge("Kingston", "Toronto");

    return myMap;
}

```

This way, we can access the nodes of the graph by their names (assuming that they are all unique names). How can we make this happen? We just need to make another **addEdge()** method that takes two **String** arguments and finds the nodes that have those labels. Perhaps we could make a nice little helper method in the **Graph** class that will find a node with a given name (label):

```

public Node nodeNamed(String aLabel) {
    for (Node n: nodes)
        if (n.getLabel().equals(aLabel)) return n;

    return null; // If we don't find one
}

```

Now we can write another `addEdge()` method that takes **String** parameters representing **Node** names:

```

public void addEdge(String startLabel, String endLabel) {
    Node start = nodeNamed(startLabel);
    Node end = nodeNamed(endLabel);

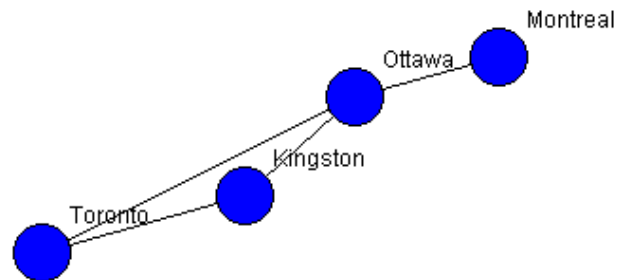
    if ((start != null) && (end != null))
        addEdge(start, end);
}

```

Notice the way we share code by making use of the "already existing" `addEdge()` method. Also notice the careful checking for valid node labels. After this new addition, the 2nd `main()` method that we created above will now work.

Displaying the Graph:

If we are going to be displaying the graph, we need to think about how we want to draw it. Here is what we "may" want to see. So where do we start? Let us work on writing code that draws each of the graph components separately.



We will start by writing methods for drawing **Nodes** and **Edges**, then use these to draw the **Graph**. We can pass around the **Graphics** object that corresponds to the "pen" that belongs to the panel. Here is a method for the **Node** class that will instruct a **Node** to draw itself using the given **Graphics** object:

```

public void draw(Graphics aPen) {
    int radius = 15;

    // Draw a blue-filled circle around the center of the node
    aPen.setColor(Color.blue);
    aPen.fillOval(location.x - radius, location.y - radius, radius*2, radius*2);

    // Draw a black border around the circle
    aPen.setColor(Color.black);
    aPen.drawOval(location.x - radius, location.y - radius, radius*2, radius*2);

    // Draw a label at the top right corner of the node
    aPen.drawString(label, location.x + radius, location.y - radius);
}

```


Notice that we draw the node twice ... once for the blue color ... once for the black border. Here is now a similar method for the **Edge** class that draws an edge:

```
public void draw(Graphics aPen) {
    // Draw black line from center of startNode to center of endNode
    aPen.setColor(Color.black);
    aPen.drawLine(startNode.getLocation().x, startNode.getLocation().y,
                  endNode.getLocation().x, endNode.getLocation().y);
}
```

When drawing the graph, we should draw edges first, then draw the nodes on top. Why not the other way around ? Here is the corresponding draw method for the **Graph** class:

```
public void draw(Graphics aPen) {
    ArrayList<Edge> edges = getEdges();

    for (Edge e: edges) // Draw the edges first
        e.draw(aPen);
    for (Node n: nodes) // Draw the nodes second
        n.draw(aPen);
}
```

The User Interface:

Now we can start the creation of our **GraphEditor** user interface. We will begin by making a panel on which we will display the graph:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class GraphEditor extends JPanel {
    private Graph aGraph; // The model (i.e. the graph)

    public GraphEditor() {
        this(new Graph());
    }
    public GraphEditor(Graph g) {
        aGraph = g;
        setBackground(Color.white);
    }

    // This is the method that is responsible for displaying the graph
    public void paintComponent(Graphics aPen) {
        super.paintComponent(aPen);
        aGraph.draw(aPen);
    }
}
```

Now we will make a class called **GraphEditorFrame** that represents a simple view which holds only our **GraphEditor** panel:

```

import javax.swing.*.*;

public class GraphEditorFrame extends JFrame {
    private GraphEditor editor;

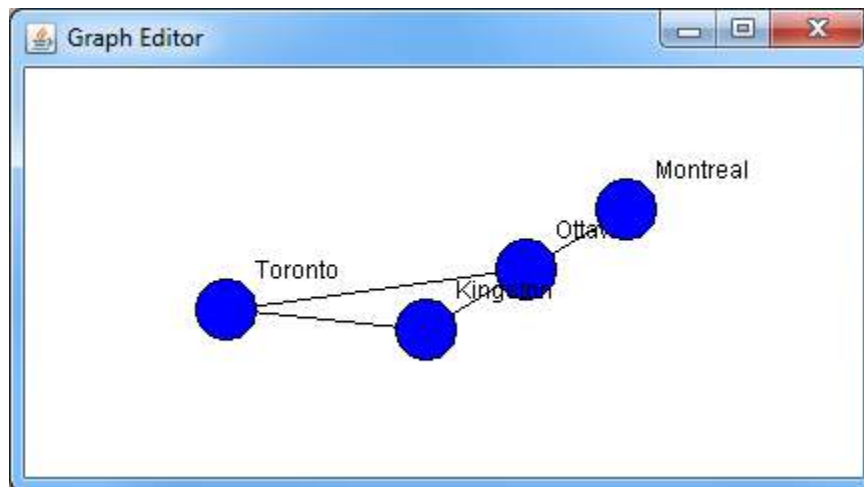
    public GraphEditorFrame (String title) {
        this(title, new Graph());
    }

    public GraphEditorFrame (String title, Graph g) {
        super(title);
        add(editor = new GraphEditor(g));
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(600, 400);
    }

    public static void main(String args[]) {
        new GraphEditorFrame("Graph Editor", Graph.example()).setVisible(true);
    }
}

```

Notice that we can run the example by running the **GraphEditorFrame** class. Our example Ontario/Quebec graph comes up right away ! This is because the `paintComponent()` method of **GraphEditor()** class is called upon startup.



Manipulating Nodes:

What kind of action should the user perform to add a node to the graph ? There are many possibilities (i.e., menu options, buttons, mouse clicks). We will allow nodes to be added to the graph via double clicks of the mouse. When the user double-clicks on the panel, a new node will be added at that click location. We must have the **GraphEditor** class implement the **MouseListener** interface. When we receive a click count of 2 on a **mouseClick** event, we will add the node at that location. For now, we will leave the other mouse listeners blank:

```

public void mouseClicked(MouseEvent event) {
    // If this was a double-click, then add a node at the mouse location
    if (event.getClickCount() == 2) {
        aGraph.addNode(new Node(event.getPoint()));
        // We have changed the model, so now we update
        update();
    }
}
public void mousePressed(MouseEvent event) { }
public void mouseReleased(MouseEvent event) { }
public void mouseEntered(MouseEvent event) { }
public void mouseExited(MouseEvent event) { }

```

Of course, we will have to add the **MouseListener** in the constructor. We will do this by calling **addEventHandlers()** which we will be adding to later on:

```

public void addEventHandlers() {
    addMouseListener(this);
}

```

```

public void removeEventHandlers() {
    removeMouseListener(this);
}

```

The **update()** method itself is quite simple since there is only one component on the window ! It merely calls **repaint()** after temporarily disabling the event handlers:

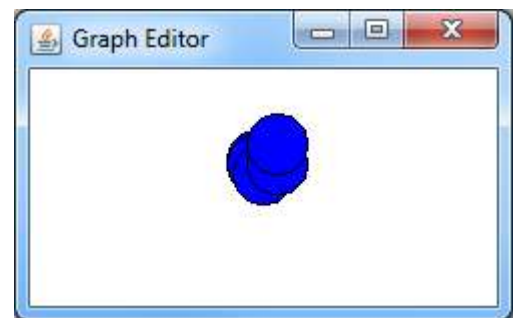
```

public void update() {
    removeEventHandlers();
    repaint();
    addEventHandlers();
}

```

If we run our code, we will notice something that is not so pleasant. Our strategy of using the double click allows us to add nodes on top of each other, making them possibly indistinguishable:

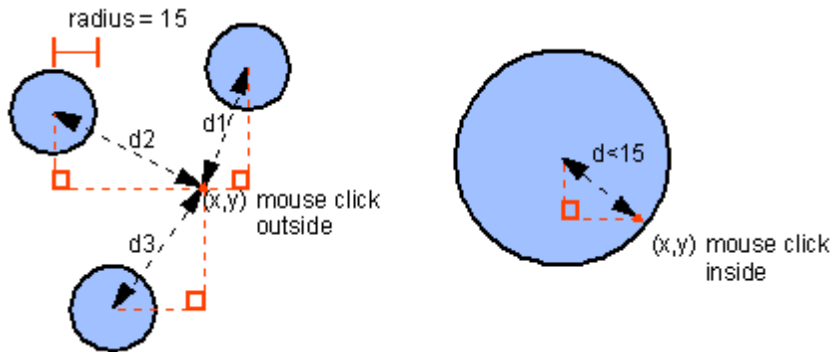
Perhaps instead of having nodes lying on top of each other, we could check to determine whether or not the user clicks within a node. Then we can decide to "not add" the node if there is already one there. What do we do then ... ignore the click ? Maybe we should cause the node to be somehow "selected" so that we can move it around.



To do this, we will need to add functionality that allows nodes to be selected and unselected.

If we attempt to re-select an "already selected" node, it should probably become unselected (i.e., toggle on/off). We should make the node appear different as well (perhaps red). We will need to detect which node has been selected. This sounds like it could be a nice little helper method in the **Graph** class.

We can just check the distance from the given point to the center of all nodes. If the distance is \leq the radius, then we are inside that node.



In fact, we are not really computing the distance, we are computing the square of the distance. This is more efficient since we do not need to compute the root. Add this to the **Graph** class:

```
// Return the first node in which point p is contained, if none, return null
public Node nodeAt(Point p) {
    for (Node n: nodes) {
        Point c = n.getLocation();
        int d = (p.x - c.x) * (p.x - c.x) + (p.y - c.y) * (p.y - c.y);
        if (d <= (15*15))
            return n;
    }
    return null;
}
```

The 15 looks like a "magic" number. It seems like this number may be used a lot. We should define a static constant in the **Node** class. Go back and change the draw method as well to use this new static value:

```
public static int    RADIUS = 15;
```

Here is the better code:

```
// Return the first node in which point p is contained, if none, return null
public Node nodeAt(Point p) {
    for (Node n: nodes) {
        Point c = n.getLocation();
        int d = (p.x - c.x) * (p.x - c.x) + (p.y - c.y) * (p.y - c.y);
        if (d <= (Node.RADIUS * Node.RADIUS)) return n;
    }
    return null;
}
```

We should go back into our drawing routines and adjust the code so that it uses this new RADIUS constant.

Now since we are allowing **Nodes** to be selected, we will have to somehow keep track of all the selected nodes. We have two choices:

- Let the graph keep track of the selected nodes separately
- Let each node keep track of whether or not it is selected

We will choose the second strategy (do you understand the tradeoffs of each?). Add the following instance variable and methods to the **Node** class:

```
private boolean    selected;
```

```
public boolean isSelected() { return selected; }
public void setSelected(boolean state) { selected = state; }
public void toggleSelected() { selected = !selected; }
```

Now we should modify the draw method to allow nodes to be selected and unselected:

```
public void draw(Graphics aPen) {
    // Draw a blue or red-filled circle around the center of the node
    if (selected)
        aPen.setColor(Color.red);
    else
        aPen.setColor(Color.blue);
    aPen.fillOval(location.x-RADIUS, location.y-RADIUS, RADIUS*2, RADIUS*2);

    // Draw a black border around the circle
    aPen.setColor(Color.black);
    aPen.drawOval(location.x-RADIUS, location.y-RADIUS, RADIUS*2, RADIUS*2);

    // Draw a label at the top right corner of the node
    aPen.drawString(label, location.x + RADIUS, location.y - RADIUS);
}
```

To make it all work, we must use it in the `mouseClicked` event handler of the **GraphEditor**:

```
public void mouseClicked(MouseEvent event) {
    // If this was a double-click, then add a node at the mouse location
    if (event.getClickCount() == 2) {
        Node    aNode = aGraph.nodeAt(event.getPoint());
        if (aNode == null)
            aGraph.addNode(new Node(event.getPoint()));
        else
            aNode.toggleSelected();

        // We have changed the model, so now we update
        update();
    }
}
```

Now how do we allow nodes to be deleted? Perhaps, the user must select the node(s) first and then hit the **delete** key. Perhaps when the **delete** key is pressed, ALL of the currently selected nodes should be deleted. So we will make a method that first returns all the selected nodes. We will need to add this method to the **Graph** class which returns a vector of all the selected nodes:

```
// Get all the nodes that are selected
public ArrayList<Node> selectedNodes() {
    ArrayList<Node> selected = new ArrayList<Node>();
    for (Node n: nodes)
        if (n.isSelected()) selected.add(n);
    return selected;
}
```

We already took care of the node selection, now we must handle the **delete** key. We should have the **GraphEditor** implement the **KeyListener** interface.

```
public void addEventHandlers() {
    addMouseListener(this);
    addKeyListener(this);
}
```

```
public void removeEventHandlers() {
    removeMouseListener(this);
    removeKeyListener(this);
}
```

```
public void keyTyped(KeyEvent event) {}

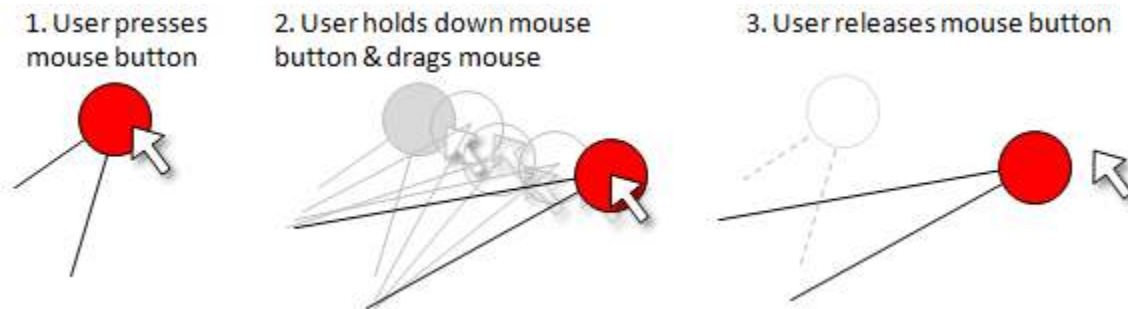
public void keyReleased(KeyEvent event) {}

public void keyPressed(KeyEvent event) {
    if (event.getKeyCode() == KeyEvent.VK_DELETE) {
        for (Node n: aGraph.selectedNodes())
            aGraph.deleteNode(n);
        update();
    }
}
```

There is a SLIGHT problem. It seems that even though we have only one component in our window (i.e., the **JPanel** which is the **GraphEditor** itself), this component does not have the focus by default. In order for the keystrokes to be detectable, the component **MUST** have the focus. So we will add the following line to the beginning of the **update()** method:

```
public void update() {
    requestFocus(); // Need this for handling KeyPress
    removeEventHandlers();
    repaint();
    addEventHandlers();
}
```

Now, how can we move nodes around once they are created? Once again, we must decide how we want the interface to work. It is most natural to allow the user to move nodes by pressing the mouse down while on top of a node and holding it down while dragging the node to the new location, then release the mouse button to cause the node to appear in the new location. We will need the **mousePressed** and **mouseDragged** events of the **MouseListener** and **MouseMotionListener** interfaces, respectively. Here is what we will have to do:



- When the user presses the mouse (i.e., a "press", not a "click"), then determine if he/she pressed on top of a node.
- If yes, then remember this node as being the one selected, otherwise do nothing
- As the mouse moves (while button being held down), we must update the chosen node's location

We will have to remember which node is being dragged so that we can keep changing its location as the mouse is dragged. We will add an instance variable in the **GraphEditor** called **dragNode** to keep this node:

```
private Node dragNode;
```

We must have the **GraphEditor** implement the **MouseEventListener** interface. Here are the updated **mousePressed** and **mouseReleased** event handlers as well as the new **mouseDragged** and **mouseMoved** event handlers which must be written:

```
// Mouse press event handler
public void mousePressed(MouseEvent event) {
    // First check to see if we are about to drag a node
    Node aNode = aGraph.nodeAt(event.getPoint());

    if (aNode != null) {
        dragNode = aNode; // If we pressed on a node, store it
    }
}
```

```
// Mouse release event handler (i.e. stop dragging process)
public void mouseReleased(MouseEvent event) {
    dragNode = null;
}
```

```
// Mouse drag event handler
public void mouseDragged(MouseEvent event) {
    if (dragNode != null)
        dragNode.setLocation(event.getPoint());
    update(); // We have changed the model, so now update
}
```

```
// Mouse drag event handler
public void mouseMoved(MouseEvent event) { /* Do Nothing */ }
```

We also need to add **addMouseMotionListener(this);** to the **addEventHandlers()** method and **removeMouseMotionListener(this);** to the **removeEventHandlers()** method. Notice that the pressing of the mouse merely stores the node to be moved. The releasing of the mouse button merely resets this stored node to **null**. All of the moving occurs in the dragging event handler. If we drag the mouse, we just make sure that we had first clicked on a node by examining the stored node just mentioned. If this stored node is not **null**, we then update its position and then update the rest of the graph. Notice that all the edges connected to a node move along with the node itself. Can you explain why ?

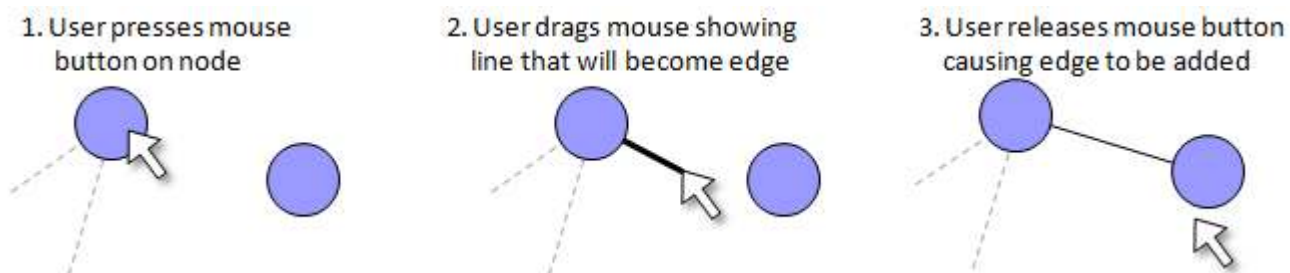
Manipulating Edges:

We have exhausted almost all the fun out of manipulating the graph nodes and we are now left with the "fun" of adding/deleting/selecting and moving edges. First we will consider adding edges. We must decide again on what action the user needs to perform in order to add the edge:

1. We can have the user double-click on the **startNode**, double click on the **endNode** and then have the edge magically appear.
2. We can select any two nodes of the graph and then perform some magic action (menu item, button press, triple click) to cause an edge to appear between the two selected edges.
3. We can click on a node and then drag the mouse to the destination node while showing the created edge as we go.

I hope you will agree that the 3rd approach is nicer in that it is more intuitive and provides the user with a nice user-friendly interface. We will see that this strategy is called **elastic banding**. To start, we will need to make the following assumptions:

- When the user presses and holds the mouse button down on a node, this node becomes the **startNode** for the edge to be created. As the user moves the mouse (i.e., **mouseDragged** event) a line should be drawn from this **startNode** to the current mouse position. When the user lets go of the mouse button on top of a different node, an edge is created between the two.
- We should abort the process of adding an edge if the user releases the mouse button while: a) not on a node or b) on the same node as he/she started.



We will have to modify the `mousePressed` , `mouseDragged` and `mouseReleased` methods.

As it turns out, the `mousePressed` event handler already stores the "start" node in the `dragNode` variable. But now look at the `mouseDragged` event handler. Currently, if we press the mouse on a node and then drag it, this will end up causing the node to be moved. But we need to allow an elastic band edge to be drawn instead of moving the node. So, we now have two behaviors that we want to do from the same action of pressing the mouse on a node. This presents a conflict since we cannot do both behaviors. Let us modify our node-moving behavior as follows:

- If the node initially clicked on is a **selected** node, then we will move it, otherwise we will assume that an edge is to be added.

The `mousePressed` event handler currently just stores the selected node. There is really nothing more to do there.

But now during the `mouseDragged` event handler, we will have to make a decision so as to either move the node (if it was a "selected" Node) or to merely draw an edge from the pressed node to the current mouse location. We cannot however, do the drawing within this method. Why ? Well, our `paintComponent()` method does the drawing and will draw over any of our drawing done here!! The drawing doesn't belong here. Drawing should happen in the `paintComponent()` method ONLY. All we will do here is just store the current mouse location in a **private Point elasticEndLocation;** variable and use it within the `paintComponent()` method.

Here are the new changes:

```
// Mouse drag event handler
public void mouseDragged(MouseEvent event) {
    if (dragNode != null) {
        if (dragNode.isSelected())
            dragNode.setLocation(event.getPoint());
    }

    // We have changed the model, so now update
    update();
}
```

Here is the updated `paintComponent()` method for the `GraphEditor` class:

```
// This is the method that is responsible for displaying the graph
public void paintComponent(Graphics aPen) {
    super.paintComponent(aPen);
    aGraph.draw(aPen);

    if (dragNode != null)
        if (!dragNode.isSelected())
            aPen.drawLine(dragNode.getLocation().x, dragNode.getLocation().y,
                elasticEndLocation.x, elasticEndLocation.y);
}
```

Notice that this method makes use of the `dragNode` and `elasticEndLocation` variables but still needs to decide whether or not to draw the elastic band line. We draw the elastic line ONLY if we are adding an edge. How do we know we are adding an edge ?

Well, we must have pressed on a starting node, so the **dragNode** must not be **null**. Also, that **dragNode** must not be selected, otherwise we are in the middle of a "node moving" operation, not an "edge adding" one.

Our last piece to this trilogy of event handler changes is to have the **mouseReleased** event handler add the new edge ONLY if we let go of the mouse button on top of a node that is not the same as the one we started with. If it is, or we let go somewhere off a node, then we must repaint everything either way to erase the elastic band:

```
// Mouse released event handler (i.e., stop dragging process)
public void mouseReleased(MouseEvent event) {
    // Check to see if we have let go on a node
    Node aNode = aGraph.nodeAt(event.getPoint());
    if ((aNode != null) && (aNode != dragNode))
        aGraph.addEdge(dragNode, aNode);

    // Refresh the panel either way
    dragNode = null;
    update();
}
```

One of our last tasks is to allow edges to be selected and removed. We can similarly add an instance variable and methods to the **Edge** class:

```
private boolean selected;
```

```
public boolean isSelected() { return selected;}

public void setSelected(boolean state) { selected = state;}

public void toggleSelected() { selected = !selected;}
```

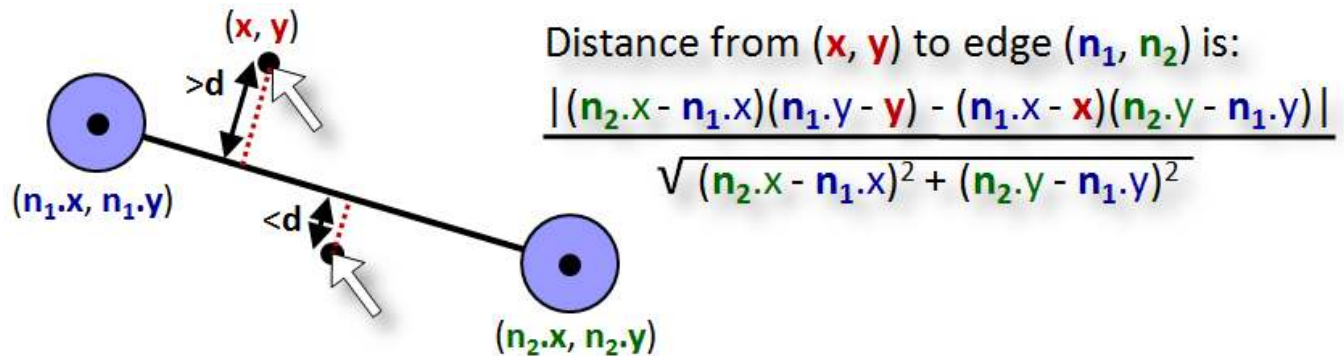
Of course ... again we must initialize the instance variable in the constructor. Now we make selected edges appear different (i.e., red).

```
// Draw the edge using the given Graphics object
public void draw(Graphics aPen) {
    // Draw black or red line from center of startNode to center of endNode
    if (selected)
        aPen.setColor(Color.red);
    else
        aPen.setColor(Color.black);

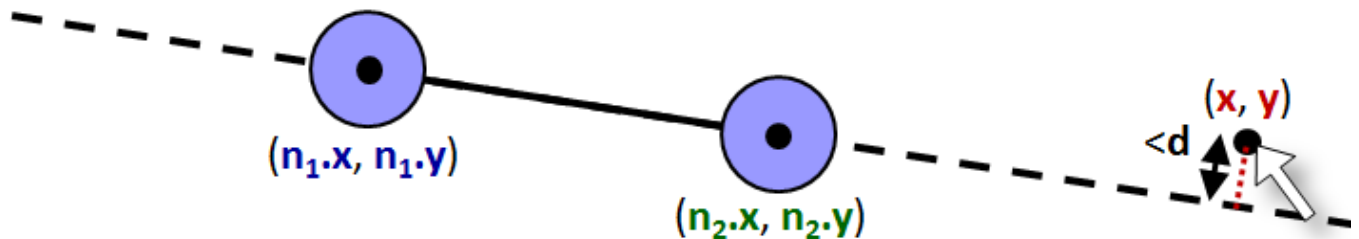
    aPen.drawLine(startNode.getLocation().x, startNode.getLocation().y,
                  endNode.getLocation().x, endNode.getLocation().y);
}
```

How does the user select an edge? Likely, by clicking on or near it. We can accomplish this by determining the distance between the point clicked at and the edge itself.

If the distance is smaller than some pre-decided value (e.g., 5 pixels) then we can assume that this edge was just clicked on... otherwise we can assume that the edge was not clicked on. The equation to find the distance from a point to an edge is indicated below:



However, the above equation actually computes the distance from (x, y) to the **line** that passes through the two edge nodes. So, if we click anywhere close to that line, we will be a small distance value and we will think that the edge was selected:



Certainly, we do not want such an (x, y) point to be considered as "close to" the edge. We can avoid this problem situation by examining the x -coordinate of the point that the user clicked on. The x -coordinate must be greater than the left node's x -coordinate and smaller than the right node's x -coordinate.

```

IF (distance < 3) THEN {
    IF ((x > n1.x) AND (x < n2.x)) OR ((x > n2.x) AND (x < n1.x)) THEN
        this edge has been selected
}

```

Can you foresee any further problems with the algorithm? What if the edge is vertical? The above checking will never select the edge! Instead, if the edge is vertical, we should compare the y -coordinates. In fact, if the line is "more horizontal" we should check the x -coordinates and if it is "more vertical" we should check the y -coordinates.

To determine if a line segment is more vertical or horizontal, we can compare the difference in x and the difference in y .

Here is the code:

```

IF (distance < 3) THEN {
    xDiff ← abs(n2.x - n1.x)
    yDiff ← abs(n2.y - n1.y)
    IF (xDiff > yDiff) THEN
        IF ((x > n1.x) AND (x < n2.x)) OR ((x > n2.x) AND (x < n1.x)) THEN
            this edge has been selected
        OTHERWISE
            IF ((y > n1.y) AND (y < n2.y)) OR ((y > n2.y) AND (y < n1.y)) THEN
                this edge has been selected
    }
}

```

Add the following method to the **Graph** class:

```

// Return first edge in which point p is near midpoint; if none, return null
public Edge edgeAt(Point p) {
    for (Edge e: getEdges()) {
        Node n1 = e.getStartNode();
        Node n2 = e.getEndNode();
        int xDiff = n2.getLocation().x - n1.getLocation().x;
        int yDiff = n2.getLocation().y - n1.getLocation().y;
        double distance = Math.abs(xDiff*(n1.getLocation().y - p.y) -
                                   (n1.getLocation().x - p.x)*yDiff) /
                           Math.sqrt(xDiff*xDiff + yDiff*yDiff);

        if (distance <= 5) {
            if (Math.abs(xDiff) > Math.abs(yDiff)) {
                if (((p.x < n1.getLocation().x) &&
                    (p.x > n2.getLocation().x)) ||
                    ((p.x > n1.getLocation().x) &&
                    (p.x < n2.getLocation().x)))
                    return e;
            }
            else
                if (((p.y < n1.getLocation().y) &&
                    (p.y > n2.getLocation().y)) ||
                    ((p.y > n1.getLocation().y) &&
                    (p.y < n2.getLocation().y)))
                    return e;
        }
    }
    return null;
}

```

Now, upon a double click, we must check for edges. We will first check to see if we clicked on a node, then if we find that we did not click on a node, we will check to see if we clicked on an edge:

```

public void mouseClicked(MouseEvent event) {
    // If this was a double-click, then add/select a node or select an edge
    if (event.getClickCount() == 2) {
        Node aNode = aGraph.nodeAt(event.getPoint());

        if (aNode == null) {
            // We missed a node, now try for an edge midpoint
            Edge anEdge = aGraph.edgeAt(event.getPoint());
            if (anEdge == null)
                aGraph.addNode(new Node(event.getPoint()));
            else
                anEdge.toggleSelected();
        }
        else
            aNode.toggleSelected();

        // We have changed the model, so now we update
        update();
    }
}

```

We can change the `keyPressed` event handler to delete all selected **Nodes AND Edges**. Of course, we will need a method to get the "selected" edges in the **Graph** class first:

```

// Get all the edges that are selected
public ArrayList<Edge> selectedEdges() {
    ArrayList<Edge> selected = new ArrayList<Edge>();
    for (Edge e: getEdges())
        if (e.isSelected()) selected.add(e);
    return selected;
}

```

```

public void keyPressed(KeyEvent event) {
    if (event.getKeyCode() == KeyEvent.VK_DELETE) {
        // First remove the selected edges
        for (Edge e: aGraph.selectedEdges())
            aGraph.deleteEdge(e);

        // Now remove the selected nodes
        for (Node n: aGraph.selectedNodes())
            aGraph.deleteNode(n);
        update();
    }
}

```

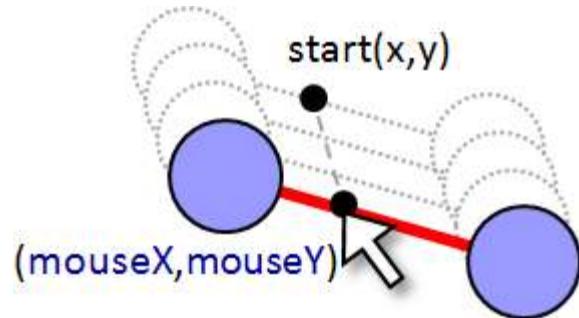
14.5 Adding Features to the Graph Editor

We have implemented a basic graph editor. There are many features that can be added. Below are solutions to some added features to the **GraphEditor**. You may want to try to add these features yourself without looking at the solutions.

Dragging Edges

- Add the following two instance variables to the **GraphEditor** class:

```
private Edge    dragEdge;
private Point   dragPoint;
```



- Add code to the **mousePressed** event handler in the **GraphEditor** class to store the edge to be dragged:

```
public void mousePressed(MouseEvent event) {
    // First check to see if we are about to drag a node
    Node    aNode = aGraph.nodeAt(event.getPoint());
    if (aNode != null) {
        // If we pressed on a node, store it
        dragNode = aNode;
        dragEdge = null;
    }
    else
        dragEdge = aGraph.edgeAt(event.getPoint());

    dragPoint = event.getPoint();
}
```

- Add code to the **mouseDragged** event handler in the **GraphEditor** class to store the edge to be dragged:

```
public void mouseDragged(MouseEvent event) {
    if (dragNode != null) {
        if (dragNode.isSelected())
            dragNode.setLocation(event.getPoint());
        else
            elasticEndLocation = event.getPoint();
    }
    if (dragEdge != null) {
        if (dragEdge.isSelected()) {
            dragEdge.getStartNode().getLocation().translate(
                event.getPoint().x - dragPoint.x, event.getPoint().y - dragPoint.y);
            dragEdge.getEndNode().getLocation().translate(
                event.getPoint().x - dragPoint.x, event.getPoint().y - dragPoint.y);
            dragPoint = event.getPoint();
        }
    }
    update(); // We have changed the model, so now update
}
```

Moving Multiple Nodes

- Add the following instance variable to the **GraphEditor** class (if not already there):

```
private Point dragPoint;
```

- Add the following line at the bottom of the **mousePressed** event handler in the **GraphEditor** class (if not already there):

```
dragPoint = event.getPoint();
```

- In the **mouseDragged** event handler for the **GraphEditor** class, change

```
if (dragNode != null) {  
    if (dragNode.isSelected())  
        dragNode.setLocation(event.getPoint());  
    else  
        elasticEndLocation = event.getPoint();  
}
```

to this:

```
if (dragNode != null) {  
    if (dragNode.isSelected()) {  
        for (Node n: aGraph.selectedNodes()) {  
            n.getLocation().translate(  
                event.getPoint().x - dragPoint.x,  
                event.getPoint().y - dragPoint.y);  
        }  
        dragPoint = event.getPoint();  
    }  
    else  
        elasticEndLocation = event.getPoint();  
}
```

Drawing Selected Edges with Different Thicknesses

- Add the following instance variable to the **Edge** class:

```
public static final int WIDTH = 7;
```

- Modify the **draw()** method in the **Edge** class:

```

public void draw(Graphics aPen) {
    if (selected) {
        aPen.setColor(Color.RED);

        int xDiff = Math.abs(startNode.getLocation().x-endNode.getLocation().x);
        int yDiff = Math.abs(startNode.getLocation().y-endNode.getLocation().y);

        for (int i= -WIDTH/2; i<=WIDTH/2; i++) {
            if (yDiff > xDiff)
                aPen.drawLine(startNode.getLocation().x+i,
                               startNode.getLocation().y,
                               endNode.getLocation().x+i,
                               endNode.getLocation().y);
            else
                aPen.drawLine(startNode.getLocation().x,
                               startNode.getLocation().y+i,
                               endNode.getLocation().x,
                               endNode.getLocation().y+i);
        }
    }
    else {
        aPen.setColor(Color.black);
        aPen.drawLine(startNode.getLocation().x, startNode.getLocation().y,
                      endNode.getLocation().x, endNode.getLocation().y);
    }
}

```

Loading and Saving Graphs

- Add the following methods to the **Node** class:

```

// Save node to given file. Note that incident edges are not saved.
public void saveTo(PrintWriter aFile) {
    aFile.println(label);
    aFile.println(location.x);
    aFile.println(location.y);
    aFile.println(selected);
}

```

```

// Load a node from given file. Note that incident edges are not connected
public static Node loadFrom(BufferedReader aFile) throws IOException {
    Node aNode = new Node();

    aNode.setLabel(aFile.readLine());
    aNode.setLocation(Integer.parseInt(aFile.readLine()),
                      Integer.parseInt(aFile.readLine()));
    aNode.setSelected(Boolean.valueOf(aFile.readLine()).booleanValue());
    return aNode;
}

```


- Add the following methods to the **Edge** class:

```
// Save edge to given file. Note that nodes themselves are not saved.
// We assume here that node locations are unique identifiers for the nodes.
public void saveTo(PrintWriter aFile) {
    aFile.println(label);
    aFile.println(startNode.getLocation().x);
    aFile.println(startNode.getLocation().y);
    aFile.println(endNode.getLocation().x);
    aFile.println(endNode.getLocation().y);
    aFile.println(selected);
}
}
```

```
// Load an edge from given file. Note that nodes themselves are not loaded.
// We are actually making temporary nodes here that do not correspond to actual
// graph nodes that this edge connects. We'll have to throw out these TEMP
// nodes later and replace them with graph nodes that connect to this edge.
public static Edge loadFrom(BufferedReader aFile) throws IOException {
    Edge    anEdge;
    String  aLabel = aFile.readLine();
    Node    start = new Node("TEMP");
    Node    end   = new Node("TEMP");

    start.setLocation(Integer.parseInt(aFile.readLine()),
                      Integer.parseInt(aFile.readLine()));
    end.setLocation(Integer.parseInt(aFile.readLine()),
                   Integer.parseInt(aFile.readLine()));

    anEdge = new Edge(aLabel, start, end);
    anEdge.setSelected(Boolean.valueOf(aFile.readLine()).booleanValue());

    return anEdge;
}
}
```

- Add the following methods to the **Graph** class:

```
// Save the graph to the given file.
public void saveTo(PrintWriter aFile) {
    aFile.println(label);

    // Output the nodes
    aFile.println(nodes.size());
    for (Node n: nodes)
        n.saveTo(aFile);

    // Output the edges
    ArrayList<Edge> edges = getEdges();
    aFile.println(edges.size());
    for (Edge e: edges)
        e.saveTo(aFile);
}
}
```

```

// Load a Graph from the given file. After the nodes and edges are loaded,
// We'll have to go through and connect the nodes and edges properly.
public static Graph loadFrom(BufferedReader aFile) throws IOException {
    // Read the label from the file and make the graph
    Graph aGraph = new Graph(aFile.readLine());

    // Get the nodes and edges
    int numNodes = Integer.parseInt(aFile.readLine());
    for (int i=0; i<numNodes; i++)
        aGraph.addNode(Node.loadFrom(aFile));

    // Now connect them with new edges
    int numEdges = Integer.parseInt(aFile.readLine());
    for (int i=0; i<numEdges; i++) {
        Edge tempEdge = Edge.loadFrom(aFile);
        Node start = aGraph.nodeAt(tempEdge.getStartNode().getLocation());
        Node end = aGraph.nodeAt(tempEdge.getEndNode().getLocation());
        aGraph.addEdge(start, end);
    }

    return aGraph;
}

```

- Change the **GraphEditorFrame** class definition to implement the **ActionListener** interface:

```
public class GraphEditorFrame extends JFrame implements ActionListener
```

- Add the following methods to the **GraphEditor** class:

```

public Graph getGraph() { return aGraph; }
public void setGraph(Graph g) { aGraph = g; update(); }

```

- Add the following to the constructor of the **GraphEditorFrame** class:

```

JMenuBar menubar = new JMenuBar();
setJMenuBar(menubar);
JMenu file = new JMenu("File");
menubar.add(file);
JMenuItem load = new JMenuItem("Load");
JMenuItem save = new JMenuItem("Save");
file.add(load);
file.add(save);
load.addActionListener(this);
save.addActionListener(this);

```

- Add the following to the **GraphEditorFrame** class (you need to **import java.io.***):

```

public void actionPerformed(ActionEvent e) {
    JFileChooser chooser = new JFileChooser(new File("."));

    if (e.getActionCommand().equals("Load")) {
        int returnVal = chooser.showOpenDialog(this);

        if (returnVal == JFileChooser.APPROVE_OPTION) {
            try {
                BufferedReader file = new BufferedReader(new FileReader(
                    chooser.getSelectedFile().getAbsolutePath()));
                editor.setGraph(Graph.loadFrom(file));
                file.close();
            }
            catch (Exception ex) {
                JOptionPane.showMessageDialog(null,
                    "Error Loading Graph From File !",
                    "Error", JOptionPane.ERROR_MESSAGE);
            }
        }
    }
    else {
        int returnVal = chooser.showSaveDialog(null);

        if (returnVal == JFileChooser.APPROVE_OPTION) {
            try {
                PrintWriter file = new PrintWriter(new FileWriter(
                    chooser.getSelectedFile().getAbsolutePath()));
                editor.getGraph().saveTo(file);
                file.close();
            }
            catch (java.io.IOException ex) {}
        }
    }
}

```

Other Features:

There are also other features we can add. Feel free to experiment with the graph editor:

- Allow all selected edges and nodes to be moved by dragging a selected edge.
- Press <CNTRL><A> to select all nodes and edges and <CNTRL><U> to unselect them.
- Right-click the mouse on a Node and prompt the user for a label to put on that node.
- Scale the entire graph up or down by holding the <SHIFT> key while pressing the mouse on an empty spot on the window and then dragging the mouse up or down to enlarge or shrink the graph.
- Press <CNTRL><D> to duplicate all selected nodes and edges and have the new portion of the graph appear a little below and to the right of the original nodes/edges.
- Show labels on edges
- Adjust labels so that they don't overlap