COMP1405/1005

Introduction to Computer Science I

Course Notes

Notes maintained by Mark Lanthier (2011 version)

Table of Contents

Chapter 1

An Introduction to Computer Science and Problem Solving

What is in This Chapter ?

This first chapter explains what computer science is all about. It will help you understand that the goal of a computer scientist is to solve problems using computers. You will see how problems are formulated by means of algorithms and how the process of abstraction can be used to break problems down to easily manageable pieces. Finally, we will discuss the notion of efficiency.

1.1 What is Computer Science ?

Computers are used just about everywhere in our society:

• Communications:	internet, e-mail, cell phones
• Word Processing:	typing/printing documents
• Business Applications:	accounting, spreadsheets
• Entertainment:	games, multimedia applications
• Database Management:	police records, stock market
• Engineering Applications:	scientific analysis, simulations
• Manufacturing:	CAD/CAM, robotics, assembly
• ... many more ...	

A computer is defined as follows (Wikipedia):

A ***computer*** *is a programmable machine that receives input, stores and manipulates data, and provides output in a useful format.*

In regards to today's computers, the "*machine*" part of the computer is called the ***hardware***, while the "*programmable*" part is called the ***software***.

Since computers are used everywhere, you can get involved with computers from just about any field of study. However, there are specific fields that are more computer-related than others. For example. the fields of ***electrical engineering*** and ***computer systems engineering*** primarily focus on the design and manufacturing of computer *hardware*, while the fields of **software engineering** and **computer science** primarily focus on the design and implementation of *software*.

Software itself can be broken down into 3 main categories:

- **System Software:** is designed to operate the computer's hardware and to provide and maintain a platform for running applications. (e.g., Windows, MacOS, Linux, Unix, etc..)

- **Middleware:** is a set of services that allows multiple processes running on one or more machines to interact. Most often used to support and simplify complex distributed applications. It can also allow data contained in one database to be accessed through another. Middleware is sometimes called *plumbing* because it connects two applications and passes data between them. (e.g., web servers, application servers).

- **Application Software:** is designed to help the user perform one or more related specific tasks. Depending on the work for which it was designed, an application can manipulate text, numbers, graphics, or a combination of these elements. (e.g., office suites, web browsers, video games, media players, etc…)

The area of software design is huge. In this course, we will investigate the basics of creating some simple application software. If you continue your degree in computer science, you will take additional courses that touch upon the other areas of system software and middleware.

Software is usually written to fulfill some need that the general public, private industry or government needs. Ideally, software is meant to make it easier for the ***user*** (i.e., the person using the software) to accomplish some task, solve some problem or entertain him/herself. Regardless of the user's motivation for using the software, many problems will arise when trying to develop the software in a way that produces correct results, is efficient ad robust, easy to use and visually appealing. That is where *computer science* comes in:

> ***Computer science*** *is the study of the theoretical foundations of information and computation, and of practical techniques for their implementation and application in computer systems (Wikipedia).*

So, computer science is all about taking in information and then performing some computations & analysis to solve a particular problem or produce a desired result, which depends on the application at hand.

Computer science is similar to mathematics in that both are used as a means of defining and solving some problem. In fact, computer-based applications often use mathematical models as a basis for the manner in which they solve the problem at hand.

In mathematics, a solution is often expressed in terms of formulas and equations. In computer science, the solution is expressed in terms of a *program*:

> *A* ***program*** *is a sequence of instructions that can be executed by a computer to solve some problem or perform a specified task.*

However, computers do not understand arbitrary instructions written in English, French, Spanish, Chinese, Arabic, Hebrew, etc..

Instead, **computers have their own languages** that they understand. Each of these languages is known as a programming language.

> *A* ***programming language*** *is an artificial language designed to automate the task of organizing and manipulating information, and to express problem solutions precisely.*

A programming language "boils down to" a set of words, rules and tools that are used to explain (or define) what you are trying to accomplish. There are many different programming languages just as there are many different "spoken" languages.

Traditional programming languages were known as ***structural*** **programming** languages (e.g., C, Fortran, Pascal, Cobol, Basic). Since the late 80's however, ***object-oriented*** **programming** languages have become more popular (e.g., JAVA, C++, C#)

There are also other types of programming languages such as ***functional*** programming languages and ***logic*** programming languages. According to the **Tiobe** index (i.e., a good site for ranking the popularity of programming languages), as of February 2011 the 10 most actively used programming languages were (in order of popularity):

Java, **C**, **C++**, **PHP**, **Python**, **C#**, **VisualBasic**, **Objective-C**, **Perl**, **Ruby**

For many years, we used JAVA as the basis in this course, due to its popularity as well as its ease of use. However, JAVA does have some drawbacks for new programmers, pertaining to some overhead in getting started with the language.

We therefore recently adjusted this course to use a language called ***Processing*** (www.processing.org) which is a JAVA-based language with much less overhead in getting started in programming. In addition, the graphical nature of the Processing language allows for more visual applications to be developed quicker and easier. You will learn more about this language as the course goes on.

When thinking of jobs and careers, many people think that computer science covers anything related to computers (i.e., anything related to *Information Technology*). However, computer science is not an area of study that pertains to IT support, repairing computers, nor installing and configuring networks. Nor does it have anything to do with simply using a computer such as doing word-processing, browsing the web or playing games. The focus of computer science is on understanding what goes on behind the software and how software/programs can be made more efficiently.

The **Computer Sciences Accreditation Board** (CSAB) identifies four general areas that it considers crucial to the discipline of computer science:

- *theory of computation*
  - investigates how specific computational problems can be solved efficiently
- *algorithms and data structures*
  - investigates efficient ways of storing, organizing and using data
- *programming methodology and languages*
  - investigates different approaches to describing and expressing problem solutions
- *computer elements and architecture*
  - investigates the design and operation of computer systems

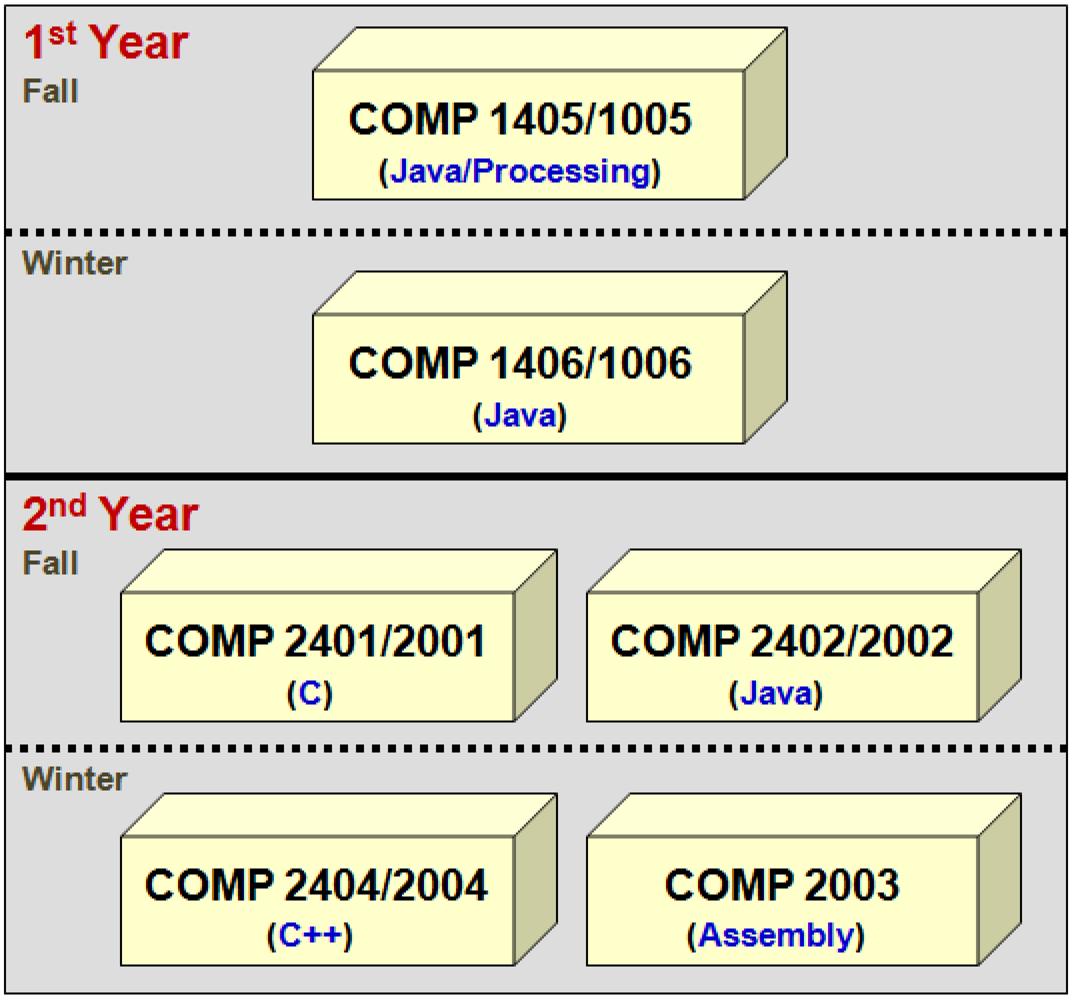
However, in addition, they also identify other important fields of computer science:

- software engineering
- artificial intelligence
- computer networking & communication
- database systems
- parallel computation
- distributed computation
- computer-human interaction
- computer graphics
- operating systems
- numerical & symbolic computation

There are aspects of each of the above fields can fall under the general areas mentioned previously. For example, within the field of **database systems** you can work on theoretical computations, algorithms & data structures, and programming methodology.

As you continue your studies in computer science, you will be able to specialize in one or more of these areas that interest you. This course, however, is meant to be an introduction to programming computers with an emphasis on problem solving.

This is your first programming course here in the School of Computer Science at Carleton. You have some more ***core*** programming courses coming up after this one. Here is a break-down of how this course fits in with your first 2 years of required programming courses:



Of course, there are other computer science courses as well. These are just the core courses that nearly everyone is required to take. After this course is over, you *should* understand how to write computer programs. In the winter term, you will take COMP1406/1006 which is a more detailed course focused on Object-Oriented programming in JAVA. Together, these two courses give you a solid programming background and you will be able to learn other computer languages easily afterwards … since they all have common features. If you want to do well in this course, attend all lectures and tutorials and do your assignments.

1.2 Writing Programs in *Processing*

It is now time to start writing simple programs to solve simple problems. As mentioned, we will be using the Processing language (available for free from www.Processing.org for your PC, MAC or Linux system).

Processing is a programming language and development environment all in one. It is an easy programming language to get started quickly in producing programs within a visual context. That means, it is a simple language that has powerful functionality for creating professional quality visual-based (i.e., graphical) applications and animations.

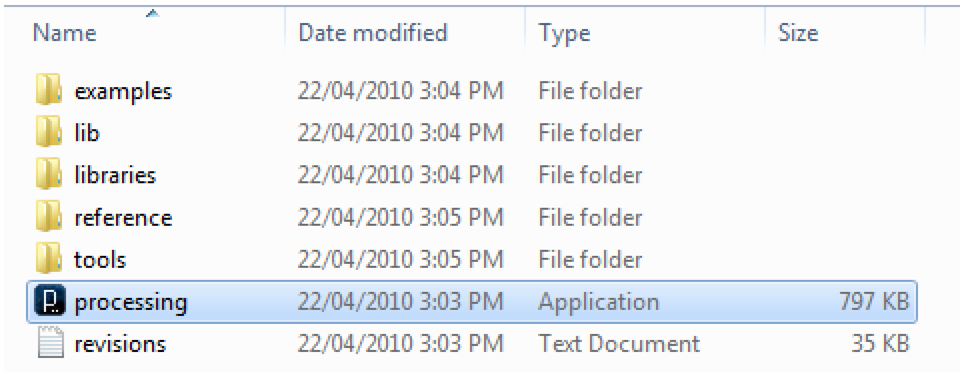
The Processing community has written over seventy libraries to help you produce applications that incorporate:

- computer vision
- data visualization
- music
- networking
- electronics

Tens of thousands of companies, artists, designers, architects, and researchers use Processing to create an incredibly diverse range of projects including:

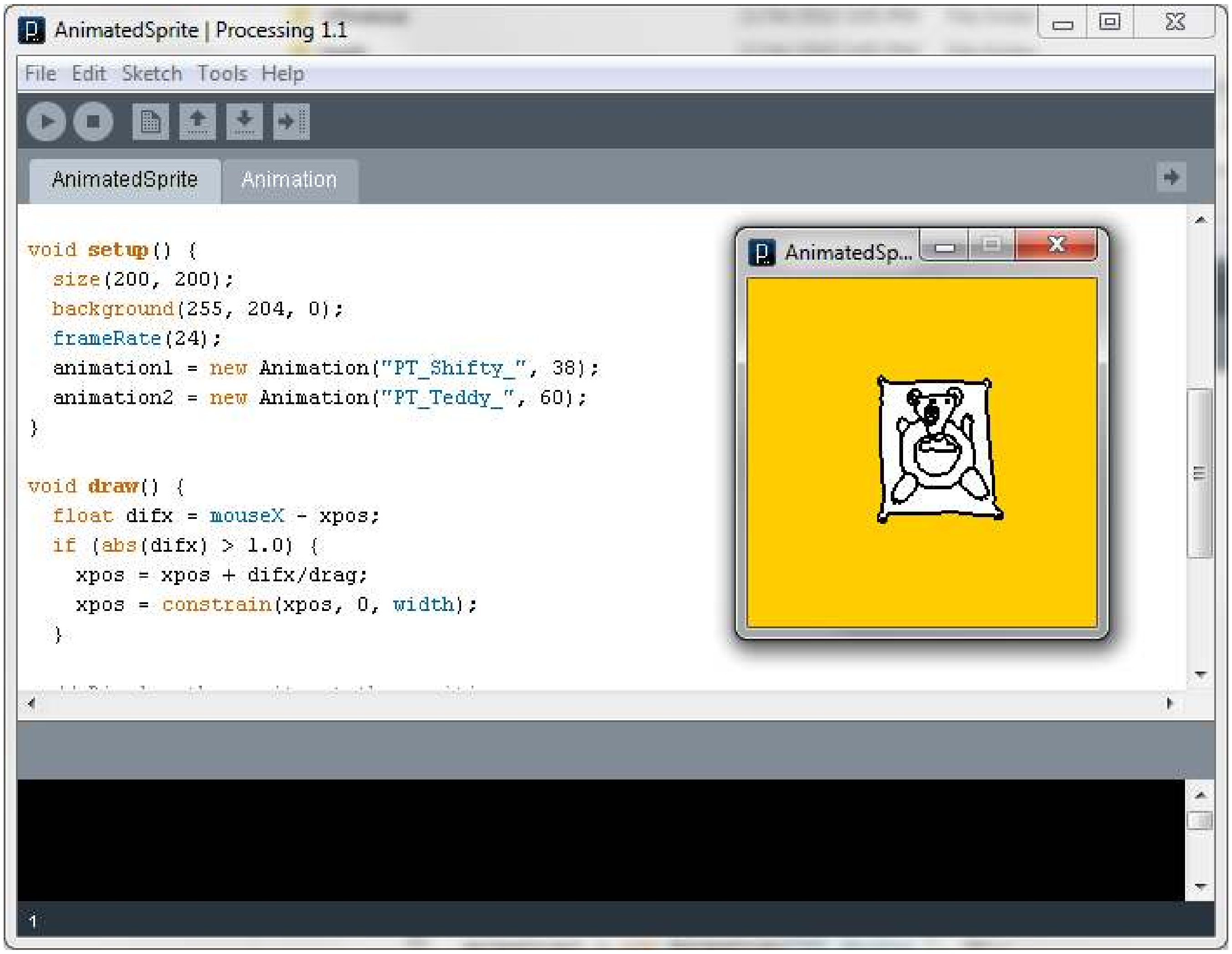
- Motion graphics for TV commercials
- Animations for music videos
- Visualizations such as that of a coastal marine ecosystem

Processing allows you to export applets for use on the web or standalone applications for the PC, Mac or Linux operating systems. To start processing, just double-click on the processing application icon that is in the processing folder that you downloaded:



Name	Date modified	Type	Size
examples	22/04/2010 3:04 PM	File folder	
lib	22/04/2010 3:04 PM	File folder	
libraries	22/04/2010 3:04 PM	File folder	
reference	22/04/2010 3:05 PM	File folder	
tools	22/04/2010 3:05 PM	File folder	
processing	22/04/2010 3:03 PM	Application	797 KB
revisions	22/04/2010 3:03 PM	Text Document	35 KB

Here is what it looks like when you are working with Processing:



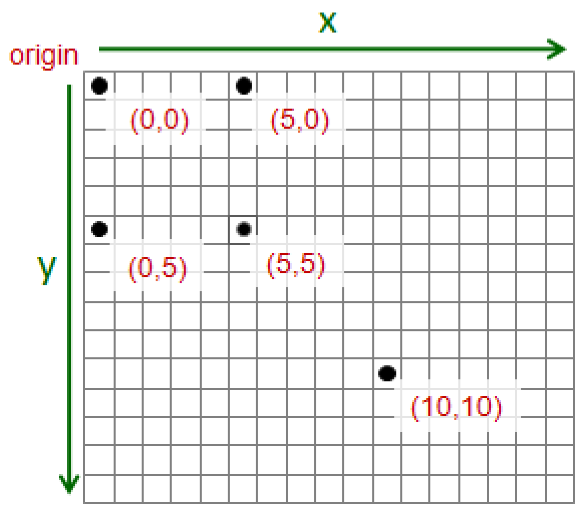
```
void setup() {
  size(200, 200);
  background(255, 204, 0);
  frameRate(24);
  animation1 = new Animation("PT_Shifty_", 38);
  animation2 = new Animation("PT_Teddy_", 60);
}

void draw() {
  float difx = mouseX - xpos;
  if (abs(difx) > 1.0) {
    xpos = xpos + difx/drag;
    xpos = constrain(xpos, 0, width);
  }
```

Each program is called a "sketch" in Processing. The top left play button starts your program which brings up a window (shown yellow here with a picture of a teddy bear). Since, many processing programs are meant to be animations, there is also a stop button beside the play button to stop the program. You should explore the Processing IDE (i.e., Integrated Development Environment) a little to get used to it.

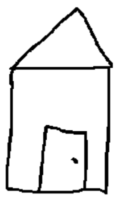
Processing uses the same syntax as JAVA. That means, Processing code looks almost exactly like JAVA code. So when you are programming in Processing, you are actually learning JAVA as well. However, Processing has been designed in a way that makes it easier to get you started because some of the overhead in getting your first program working is hidden.

As you may recall, Processing is a graphics-based language and therefore we will spend a lot of time and effort drawing various things on the screen. When drawing anything, it is important to specify **where** you want to draw.

The output screen is organized as a 2-dimensional (2D) grid of pixels organized by the standard **x** and **y** coordinate system. That is, given an **(x,y)** pair, which we call a ***point***, the **x** specifies a number of horizontal pixels from the ***origin*** (or *start location* at the top-left of the screen) while **y** specifies the number of vertical pixels from the origin. So, point **(0,0)** is the origin and is at the top-left of the screen.



Lets write our first program. Lets draw a simple house like this one shown here. This involves drawing a square, a triangle, a rectangle and a dot.



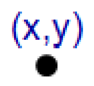
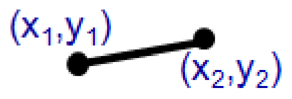
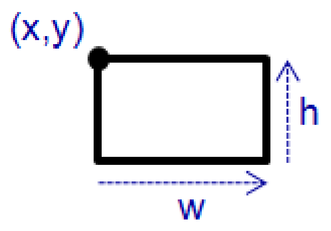
Since Processing is a graphical-based language, there are pre-defined functions for drawing shapes. Each of these functions requires some ***parameters*** to specify further information about how to do the drawing such as locations and dimensions of what we are trying to draw.

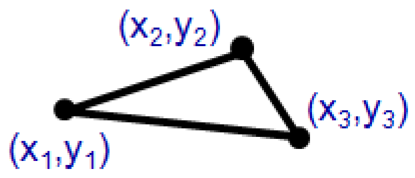
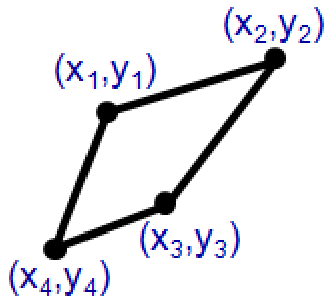
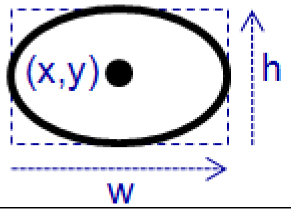
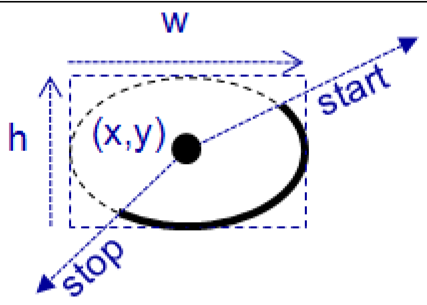
Looking at the table of functions on the next page, it should be clear that we need to call the following functions in order to draw our house:

- **rect(x, y, w, h)** – for the main frame
- **triangle(x_1, y_1, x_2, y_2, x_3, y_3)** – for the roof
- **rect(x, y, w, h)** – for the door
- **point(x, y)** – for the door handle

All we need to do then is to figure out what the parameters should be.

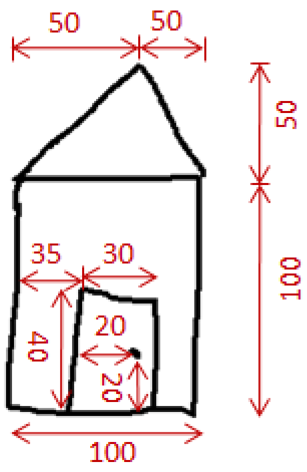
Here are the ones that we can use to draw 2-dimensional shapes:

Function	Description	Example
point(x,y)	draws a single **dot** at the location specified by **x** and **y**.	
line(x_1,y_1,x_2,y_2)	draws a **line** from location (**x_1, y_1**) to location (**x_2, y_2**).	
rect(x,y,w,h)	draws a **rectangle** with its top-left at location (**x, y**). The width and height of the rectangle are **w** and **h**. If **w** and **h** are equal, a **square** is drawn.	

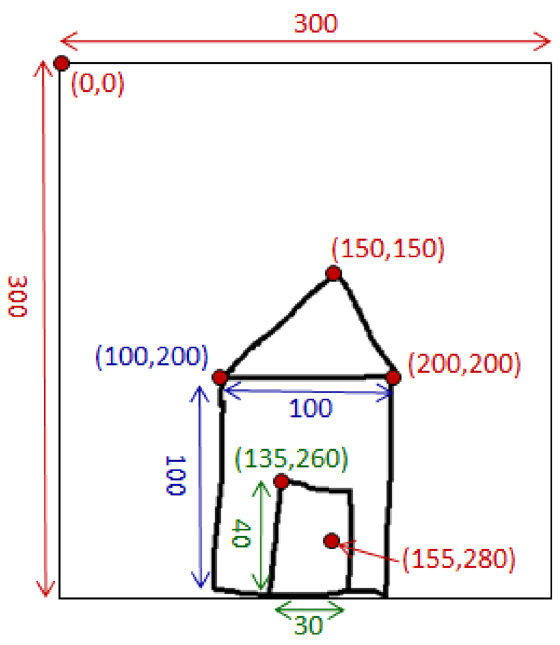
triangle(x_1,y_1,x_2,y_2,x_3,y_3)	draws 3 lines in order from (**x_1, y_1**) to (**x_2, y_2**) to (**x_3, y_3**) and back to (**x_1, y_1**) to form a **triangle**.	
quad(x_1,y_1,x_2,y_2,x_3,y_3,x_4,y_4)	draws 4 lines in order from (**x_1, y_1**) to (**x_2, y_2**) to (**x_3, y_3**) to (**x_4, y_4**) and back to (**x_1, y_1**) to form a **4-sided shape**.	 (x_2,y_2) (x_1,y_1) (x_3,y_3) (x_4,y_4)
ellipse(x,y,w,h)	draws an **ellipse** (or **oval**) with its *center* at location (**x, y**). The width and height of the ellipse are **w** and **h**. If **w** and **h** are equal, a **circle** is drawn.	
arc(x,y,w,h,start,stop)	draws an **arc** (i.e., a **portion of an ellipse or circle**) with its *center* at location (**x, y**). The width and height of the ellipse are **w** and **h**. If **w** and **h** are equal, a **circle** is drawn. The arc is draw from the **start** number of radians to the **stop** number of radians.	

So the first step is to figure out the dimensions of the house in pixels. To do this, we begin with our hand-drawn sketch. Then, we add some dimensions to the house (in pixels).

Here to the right, is one possible set of dimensions for the house. Note that the drawing is quite rough and so the dimensions are not necessarily to scale.



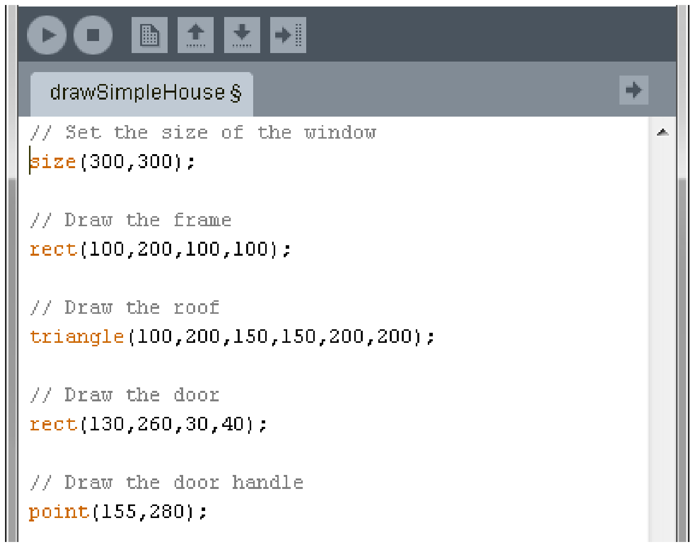
Next, we'll need to know roughly how big the drawing area will be. Sometimes it is a good idea to know this even before we decide upon the dimensions so that our house is the "proper" size according to the context of the drawing area. Assume that we choose a drawing area of **300 x 300** pixels.



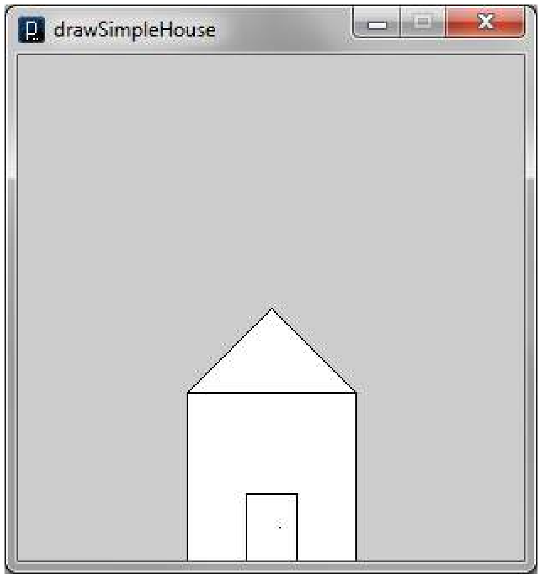
Now we need to decide *where* within the area the house will be located and assign point values to the corners of the house … remembering that (0,0) is at the top left corner of the drawing area.

Note that for rectangles, we just need to know the top left corner, along with the dimensions.

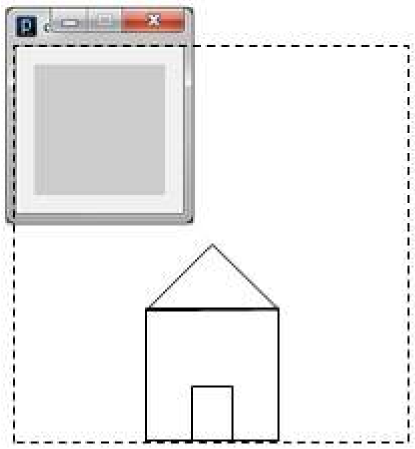
Here is the final program:



```
// Set the size of the window
size(300,300);

// Draw the frame
rect(100,200,100,100);

// Draw the roof
triangle(100,200,150,150,200,200);

// Draw the door
rect(130,260,30,40);

// Draw the door handle
point(155,280);
```



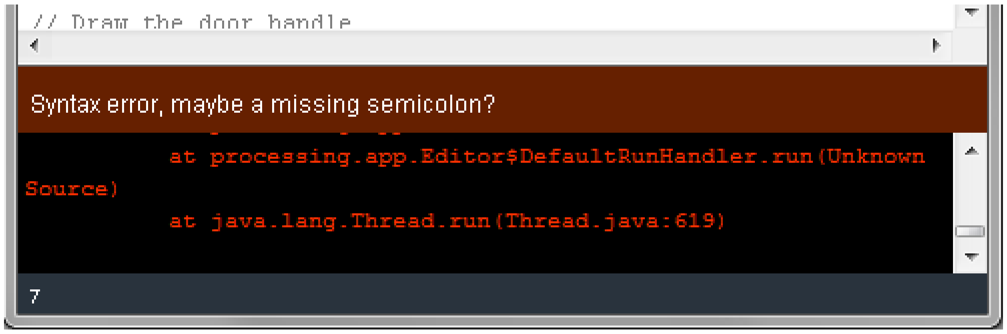
The **size(w,h)** function in Processing allows you to specify the size of the drawing area. If you do not call this method, the default window size will be **100 x 100**.



In our case, if we did not call **size(300,300)**, then a small window would appear and the house would not be shown since it would be drawn off the window's visible area as shown here:

"Drawing things off the screen" is a common error that many students encounter when writing graphical programs.

So … we have just created our first Processing program. Notice that each function call ends with a **;** character. That's how you tell Processing (and JAVA) that you are done that step of the program. It is like having a period at the end of an English sentence to indicate that the sentence has ended and a new one is about to begin. Leaving off a semi-colon somewhere is considered a syntax error (i.e., like a spelling or grammar error) and will prevent your program from running:



```
// Draw the door handle
```

Syntax error, maybe a missing semicolon?

```
        at processing.app.Editor$DefaultRunHandler.run(Unknown
Source)
        at java.lang.Thread.run(Thread.java:619)
```

7

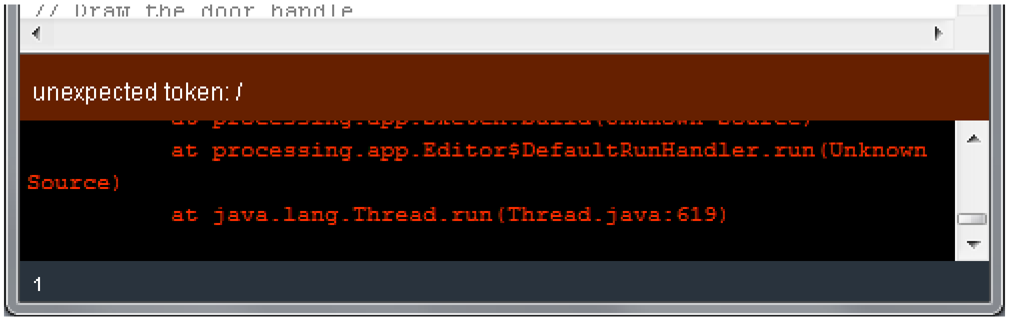
If this happens, Processing usually highlights the line in your program that appears just after the line that is missing the semicolon.

You may also notice that there are lines with double slashes ***//***. These are called ***comments***. It tells the Processing interpreter that the rest of the line is to be ignored by the program. This allows you to put English-like explanations throughout your code so that anyone who read your program later (possibly you yourself) will understand what you are doing in that part of the program. It is always a good idea to use comments but not too many of them such that you code becomes too cluttered. Use just enough to make it clear what you are doing in that part of your program. Throughout the course, take notice of *where* and *how many* comments are being used in the example programs that we do together.

You can also make multiple-line comments by beginning with ***/**** and ending with ****/***. For example, this could be a comment at the top of your program:

```
/* DrawSimpleHouse:
   ---------------
   This program draws a simple house that has
   A frame, a roof and a door with a door knob. */
```

Forgetting one / in your comment will produce this error:



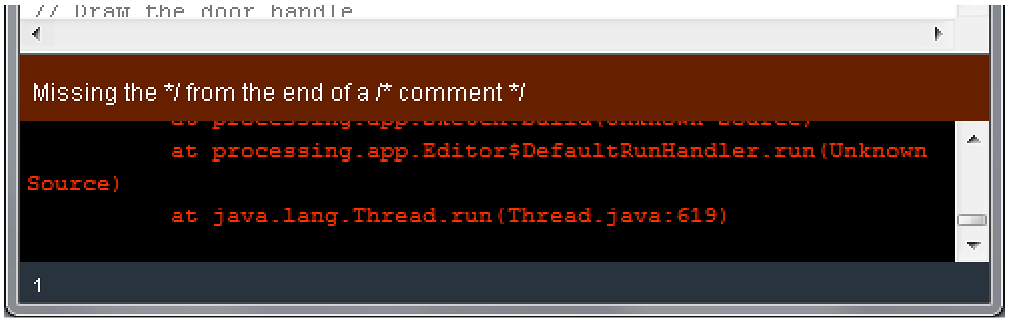
```
// Draw the door handle
```

unexpected token: /

```
        at processing.app.Editor$DefaultRunHandler.run(Unknown
Source)
        at java.lang.Thread.run(Thread.java:619)
```

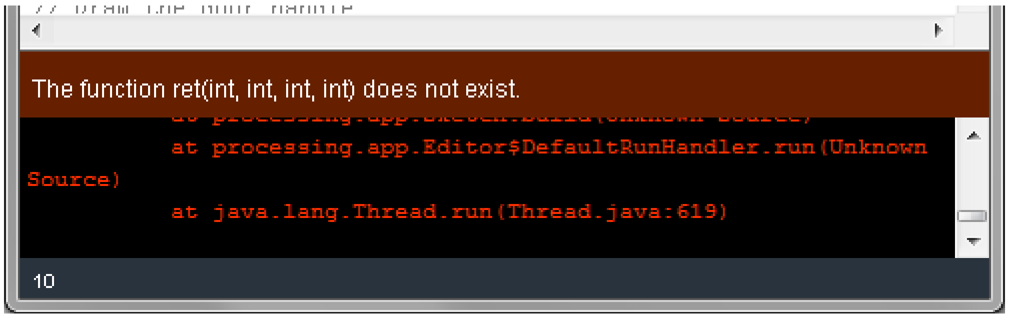
1

And forgetting to close a multi-line comment will produce this error:



```
// Draw the door handle
Missing the */ from the end of a /* comment */
        at processing.app.Editor$DefaultRunHandler.run(Unknown
Source)
        at java.lang.Thread.run(Thread.java:619)
1
```

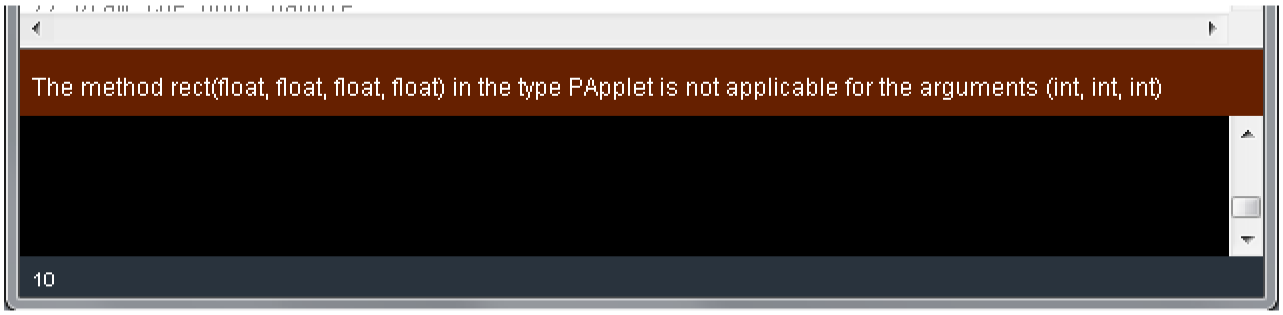
As you learn to program, you will find many more errors. For example, a VERY common error is to spell one of the function names wrong. If you accidentally spell **rect** as **ret**, for example, this would be the error that you get:



```
// Draw the door handle
The function ret(int, int, int, int) does not exist.
        at processing.app.Editor$DefaultRunHandler.run(Unknown
Source)
        at java.lang.Thread.run(Thread.java:619)
10
```

You will notice however, that when you spell a function name wrong, you usually notice it because it will not appear with the proper "orange" color that all the other functions have.

A less obvious error message occurs when you miss one of the parameters, or add too many, or pass the wrong type of values to the function. For example, assume that you called **rect(100, 200, 100)** instead of **rect(100, 200, 100, 100)**. This is the "less understandable" error that you would get:



```
The method rect(float, float, float, float) in the type PApplet is not applicable for the arguments (int, int, int)
10
```

There are many more functions that you can play around with.

When using computers, colors are often represented as:

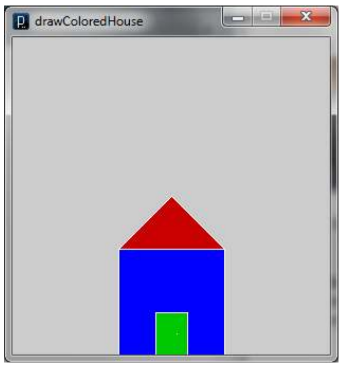
- ***Grayscale***: a shade of gray value (from **0 to 255**) where **0** represents **black**, **255** represents **white** and values in between represent various levels of gray.

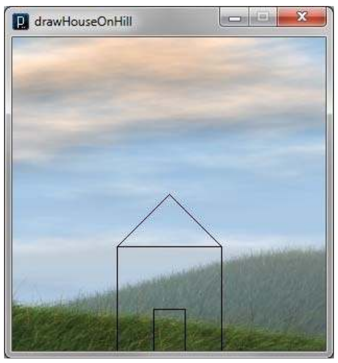
- ***RGB***: 3 values (from **0 to 255**) representing the amount of red, green and blue in the color. Bright **red**, for example would be represented by parameters (**255, 0, 0**), bright **green** as (**0, 255, 0**) and royal **blue** as (**0, 0, 255**). By varying these values, you can obtain any color of the rainbow.

Here are a few color-related functions that you can make use of:

Function	**Description**
stroke(gray);	Sets the border-color for lines & shapes (e.g., points, lines, arcs, rectangles, triangles, ellipses) to a shade of gray specified by the value of the **gray** parameter.
stroke(r, g, b);	Sets the border-color for shapes to the color with the given amount of red, green and blue, specified by parameters **r**, **g** and **b**, respectively.
noStroke();	Sets the border-color to be transparent (i.e., glass).
fill(gray);	Sets the fill-in-color for enclosed shapes (e.g., rectangles, triangles, ellipses) to a shade of gray specified by the value of the **gray** parameter.
fill(r, g, b);	Sets the fill-in-color for enclosed shapes to the color with the given amount of red, green and blue, specified by parameters **r**, **g** and **b**, respectively.
noFill();	Sets the fill-in-color to be transparent (i.e., glass).
background(gray);	Sets the drawing area's background to a shade of gray specified by the value of the **gray** parameter.
background (r, g, b);	Sets the drawing area's background to the color with the given amount of red, green and blue, specified by parameters **r**, **g** and **b**, respectively.
background(**loadImage**("hills.png"));	Sets the drawing area's background to the image specified by the given image file name which may be a **png**, **gif** or **jpg** file.

See if you can produce the following images (of course, you would need to supply your own 300 x 300 image for the example with the background hills):

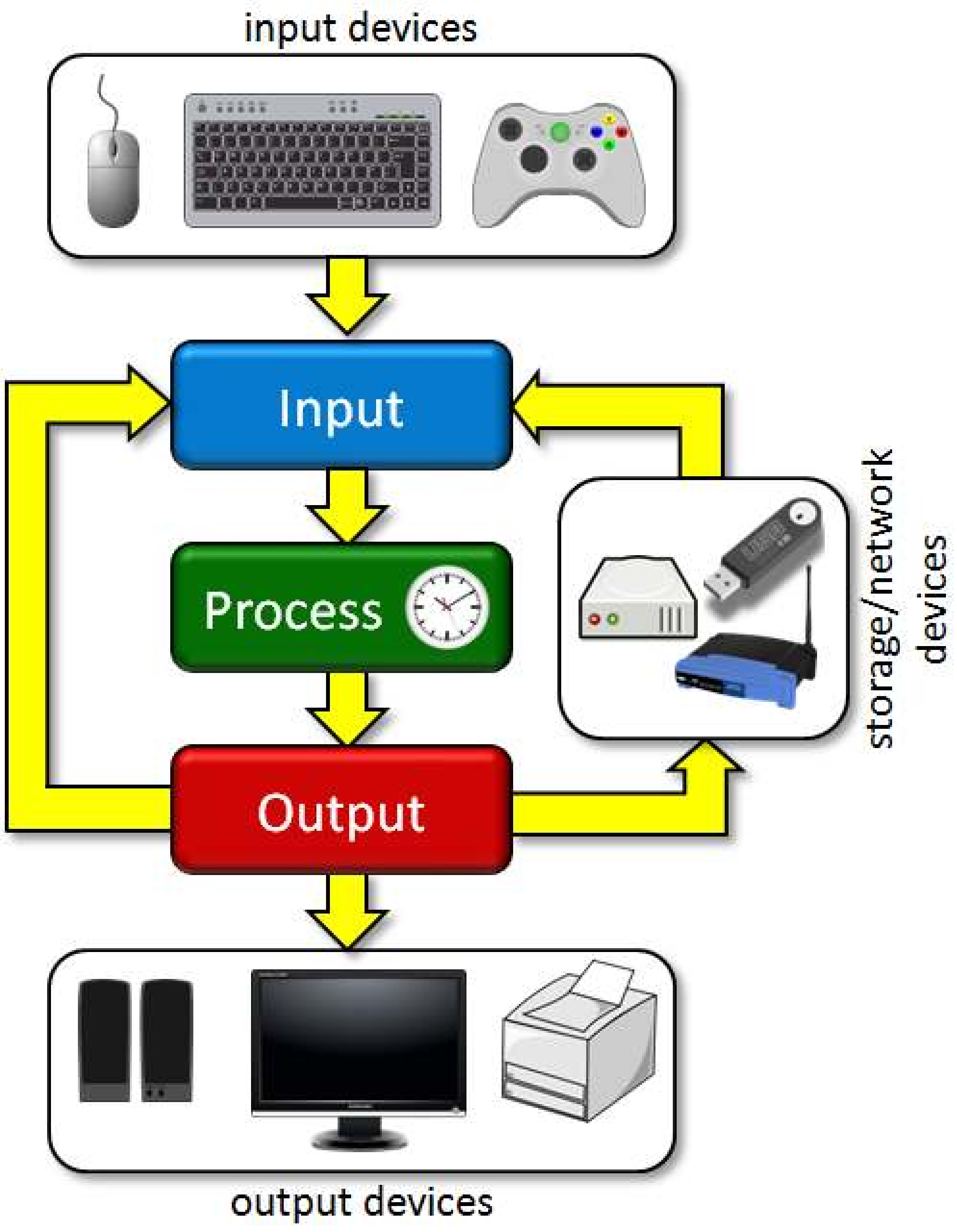




1.3 Problem Solving

Regardless of the area of study, computer science is all about solving problems with computers. The problems that we want to solve can come from any real-world problem or perhaps even from the abstract world. We need to have a standard systematic approach to solving problems.

Since we will be using computers to solve problems, it is important to first understand the computer's information processing model. The model shown in the diagram below assumes a single CPU (Central Processing Unit). Many computers today have multiple CPUs, so you can imagine the above model duplicated multiple times within the computer.



A typical single CPU computer processes information as shown in the diagram. Problems are solved using a computer by obtaining some kind of user input (e.g., keyboard/mouse information or game control movements), then processing the input and producing some kind of output (e.g., images, test, sound). Sometimes the incoming and outgoing data may be in the form of hard drives or network devices.

In regards to problem solving, we will apply the above model in that we will assume that we are given some kind of input information that we need to work with in order to produce some desired output solution. However, the above model is quite simplified. For larger and more complex problems, we need to iterate (i.e., repeat) the input/process/output stages multiple times in sequence, producing intermediate results along the way that solve part of our problem, but not necessarily the whole problem. For simple computations, the above model is sufficient.

It is the "problem solving" part of the process that is the interesting part, so we'll break this down a little. There are many definitions for "problem solving". Here is one:

> ***Problem Solving*** *is the sequential process of analyzing information related to a given situation and generating appropriate response options.*

There are 6 steps that you should follow in order to solve a problem:

1. Understand the Problem
2. Formulate a Model
3. Develop an Algorithm
4. Write the Program
5. Test the Program
6. Evaluate the Solution

Consider a simple example of how the input/process/output works on a simple problem:

Example: *Calculate the average grade for all students in a class.*

1. **Input**: get all the grades … perhaps by typing them in via the keyboard or by reading them from a USB flash drive or hard disk.

2. **Process**: add them all up and compute the average grade.

3. **Output**: output the answer to either the monitor, to the printer, to the USB flash drive or hard disk … or a combination of any of these devices.

As you can see, the problem is easily solved by simply getting the input, computing something and producing the output. Let us now examine the 6 steps to problems solving within the context of the above example.

STEP 1: Understand the Problem:

It sounds strange, but the first step to solving any problem is to make sure that you understand the problem that you are trying to solve. You need to know:

- What input data/information is available ?
- What does it represent ?
- What format is it in ?
- Is anything missing ?
- Do I have everything that I need ?
- What output information am I trying to produce ?
- What do I want the result to look like … text, a picture, a graph … ?
- What am I going to have to compute ?

In our example, we well understand that the input is a bunch of grades. But we need to understand the **format** of the grades. Each grade might be a **number from 0 to 100** or it may be a **letter grade from A+ to F**. If it is a number, the grade might be a **whole integer** like 73 or it may be a **real number** like 73.42. We need to understand the format of the grades in order to solve the problem.

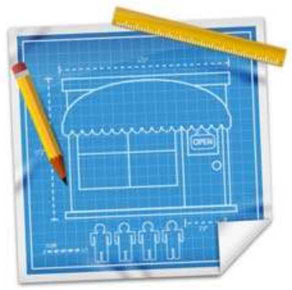
We also need to consider missing grades. What if we do not have the grade for **every** student (e.g., some were away during the test) ? Do we want to be able to **include** that person in our average (i.e., they received 0) or **ignore** them when computing the average ?

We also need to understand what the output should be. Again, there is a formatting issue. Should we output a **whole** or **real number** or a **letter grade** ? Maybe we want to display a pie chart with the average grade. It is our choice.

Finally, we should understand the kind of processing that needs to be performed on the data. This leads to the next step.

STEP 2: Formulate a Model:

Now we need to understand the processing part of the problem. Many problems break down into smaller problems that require some kind of simple mathematical computations in order to process the data. In our example, we are going to compute the average of the incoming grades. So, we need to know the model (or formula) for computing the average of a bunch of numbers. If there is no such "formula", we need to develop one. Often, however, the problem breaks down into simple computations that we well understand. Sometimes, we can look up certain formulas in a book or online if we get stuck.



In order to come up with a model, we need to fully understand the information available to us. Assuming that the input data is a bunch of integers or real numbers $\mathbf{x_1,x_2,\ldots,x_n}$ representing a grade percentage, we can use the following computational model:

$$\mathbf{Average1 = (x_1 + x_2 + x_3 + \ldots + x_n) / n}$$

where the result will be a number from 0 to 100.

That is very straight forward (assuming that we knew the formula for computing the average of a bunch of numbers). However, this approach will not work if the input data is a set of letter grades like **B-, C, A+, F, D-**, etc.. because we cannot perform addition and division on the letters. This problem solving step must figure out a way to produce an average from such letters. Thinking is required.

After some thought, we may decide to assign an integer number to the incoming letters as follows:

A^+ = 12	**B^+** = 9	**C^+** = 6	**D^+** = 3	**F** = 0
A = 11	**B** = 8	**C** = 5	**D** = 2	
A^- = 10	**B^-** = 7	**C^-** = 4	**D^-** = 1	

If we assume that these newly assigned grade numbers are $\mathbf{y_1,y_2,\ldots,y_n}$, then we can use the following computational model:

$$\mathbf{Average2 = (y_1 + y_2 + y_3 + \ldots + y_n) / n}$$

where the result will be a number from 0 to 12.

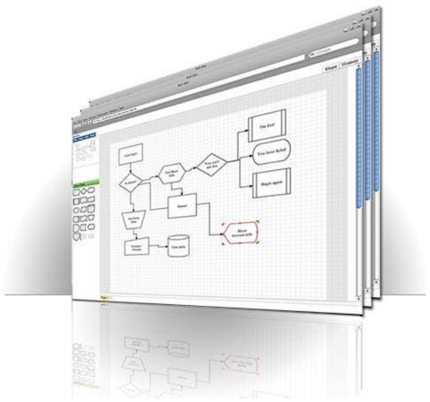
As for the output, if we want it as a percentage, then we can use either **Average1** directly or use **(Average2 / 12)**, depending on the input that we had originally. If we wanted a letter grade as output, then we would have to use **(Average1/100*12)** or **(Average1*0.12)** or **Average2** and then map that to some kind of "lookup table" that allows us to look up a grade letter according to a number from 0 to 12.

Do you understand this step in the problems solving process ? It is all about figuring out how you will make use of the available data to compute an answer.

STEP 3: Develop an Algorithm:

Now that we understand the problem and have formulated a model, it is time to come up with a precise plan of what we want the computer to do.

An algorithm is a precise sequence of instructions for solving a problem.

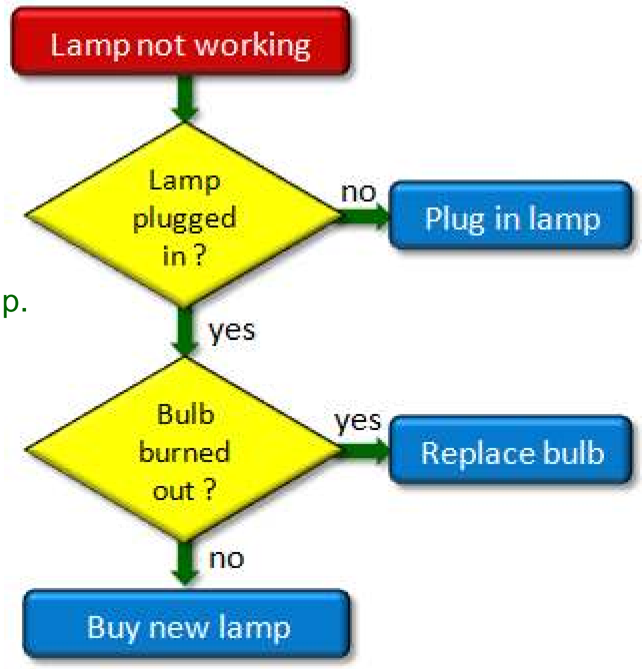


Some of the more complex algorithms may be considered "**randomized algorithms**" or "**non-deterministic algorithms**" where the instructions are not necessarily in sequence and in may not even have a finite number of instructions. However, the above definition will apply for all algorithms that we will discuss in this course.

To develop an algorithm, we need to represent the instructions in some way that is understandable to a person who is trying to figure out the steps involved. Two commonly used representations for an algorithm is by using (1) ***pseudo code***, or (2) ***flow charts***. Consider the following example (from Wikipedia) of solving the problem of a broken lamp. To the right is an example of a flow chart, while to the left is an example of pseudocode for solving the same problem:

Pseudo Code

```
1. IF lamp works, go to step 7.
2. Check if lamp is plugged in.
3. IF not plugged in, plug in lamp.
4. Check if bulb is burnt out.
5. IF blub is burnt, replace bulb.
6. IF lamp doesn't work buy new lamp.
7. Quit ... problem is solved.
```



Notice that:

pseudocode is a simple and concise sequence of English-like instructions to solve a problem.

Pseudocode is often used as a way of describing a computer program to someone who doesn't understand how to program a computer. When learning to program, it is important to write pseudocode because it helps you clearly understand the problem that you are trying to solve. It also helps you avoid getting bogged down with syntax details (i.e., like spelling mistakes) when you write your program later (see step 4).

Although flowcharts can be visually appealing, pseudocode is often the preferred choice for algorithm development because:

- It can be difficult to draw a flowchart neatly, especially when mistakes are made.
- Pseudocode fits more easily on a page of paper.
- Pseudocode can be written in a way that is very close to real program code, making it easier later to write the program (i.e., in step 4).
- Pseudocode takes less time to write than drawing a flowchart.

Pseudocode will vary according to whoever writes it. That is, one person's pseudocode is often quite different from that of another person. However, there are some common control structures (i.e., features) that appear whenever we write pseudocode.

These are shown here along with some examples:

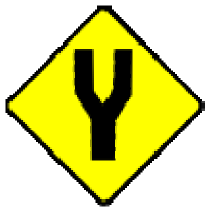
- **sequence**: listing instructions step by step in order (often numbered)

```
1. Make sure switch is turned on
2. Check if lamp is plugged in
3. Check if bulb is burned out
4. …
```

- **condition**: making a decision and doing one thing or something else depending on the outcome of the decision.

```
if lamp is not plugged in
   then plug it in
```

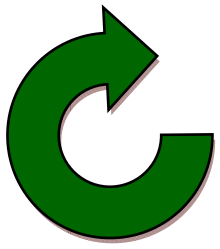
```
if bulb is burned out
   then replace bulb
otherwise buy new lamp
```



- **repetition**: repeating something a fixed number of times or until some condition occurs.

```
repeat
   get a new light bulb
   put it in the lamp
until lamp works or no more bulbs left
```

```
repeat 3 times
   unplug lamp
   plug into different socket
…
```

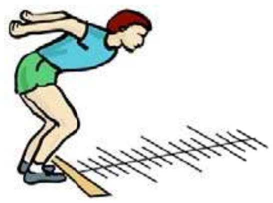


- **storage**: storing information for use in instructions further down the list

```
x ← a new bulb
count ← 8
```

- **jumping**: being able to jump to a specific step when needed

```
if bulb works
   then goto step 7
```



You will notice that the bold in the above examples highlights the specific control structure. You will notice that for the condition and repetition structures, the portion of the pseudocode that is part of the condition or the repeat loop are indented a bit so as to make it clear that these are kinds "inner steps" that belong to that structure. Some people will us brackets to indicate what is in or out of a control structure as follows:

```
if (bulb is burned out) then {
   replace bulb
}
otherwise {
   buy new lamp
}
```

```
repeat {
    get a new light bulb
    put it in the lamp
} until (lamp works or no more bulbs left)
```

```
repeat 3 times {
    unplug lamp
    plug into different socket
}
```

The point is that there are a variety of ways to write pseudocode. The important thing to remember is that your algorithm should be clearly explained with no ambiguity as to what order your steps are performed in.

Whether using a flow chart of pseudocode, you should test your algorithm by manually going through the steps in your head to make sure that you did not forget a step or a special situation. Often, you will find a flaw in your algorithm because you forgot about a special situation that could arise. Only when you are convinced that your algorithm will solve your problem, should you go ahead to the next step.

Consider our previous example of finding the average of a set of **n** grades stored in a file. What would the pseudocode look like ? Here is an example of what it might look like if we had the example of **n** numeric grades $\mathbf{x_1 \ldots x_n}$ that were loaded from a file:

Algorithm: DisplayGrades

```
1.      set the sum of the grade values to 0.
2.      load all grades $x_1 \ldots x_n$ from file.
3.      repeat n times {
4.              get grade $x_i$
5.              add $x_i$ to the sum
        }
6.      compute the average to be sum / n.
7.      print the average.
```

It would be wise to run through the above algorithm with a real set of numbers. Each time we test an algorithm with a fixed set of input data, this is known as a ***test case***. You can create many test cases. Here are some to try:

n = 5, x_1 = 92, x_2 = 37, x_3 = 43, x_4 = 12, x_5 = 71 … result should be **51**
n = 3, x_1 = 1, x_2 = 1, x_3 = 1 …………………………... result should be **1**
n = 0 ……………………………………………………… result should be **0**

STEP 4: Write the Program:

Now that we have a precise set of steps for solving the problem, most of the hard work has been done. We now have to transform the algorithm from step 3 into a set of instructions that can be understood by the computer.

Writing a program is often called "***writing code***" or "***implementing an algorithm***". So the ***code*** (or ***source code***) is actually the program itself.

Without much of an explanation, below is a program (written in processing) that implements our algorithm for finding the average of a set of grades. Notice that the code looks quite similar in structure, however, the processing code is less readable and seems somewhat more mathematical:

Pseudocode	Processing code (i.e., program)

1. set the sum of the grade values to 0.
2. load all grades x_1 **…** x_n from file.
3. **repeat n times** {
4. get grade x_i
5. add x_i to the sum
}
6. compute the average to be sum / **n**.
7. print the average.
	`int sum = 0;`
	`byte[] x = loadBytes("numbers");`
	`for (int i=0; i<x.length; i++)`
	`  sum = sum + x[i];`
	
	`int avg = sum / x.length;`
	`print(avg);`

For now, we will not discuss the details of how to produce the above source code. In fact, the source code would vary depending on the programming language that was used. Learning a programming language may seem difficult at first, but it will become easier with practice.

The computer requires precise instructions in order to understand what you are asking it to do. For example, if you removed one of the semi-colon characters (**;**) from the program above, the computer would become confused as to what you are doing because the (;) is what it understands to be the end of an instruction. Leaving one of them off will cause your program to generate what is known as a ***compile error***.

Compiling is the process of converting a program into instructions that can be understood by the computer.

The longer your program becomes, the more likely you will have multiple compile errors. You need to fix all such compile errors before continuing on to the next step.

STEP 5: Test the Program:

Once you have a program written that compiles, you need to make sure that it solves the problem that it was intended to solve and that the solutions are correct.

Running a program is the process of telling the computer to evaluate the compiled instructions.

When you run your program, if all is well, you should see the correct output. It is possible however, that your program works correctly for some set of data input but not for all. If the output of your program is incorrect, it is possible that you did not convert your algorithm properly into a proper program. It is also possible that you did not produce a proper algorithm back in step 3 that handles all situations that could arise. Maybe you performed some instructions out of sequence. Whatever happened, such problems with your program are known as ***bugs***.

Bugs are problems/errors with a program that cause it to stop working or produce incorrect or undesirable results.

You should fix as many bugs in your program as you can find. To find bugs effectively, you should test your program with many test cases (called a ***test suite***). It is also a good idea to have others test your program because they may think up situations or input data that you may never have thought of. The process of finding and fixing errors in your code is called ***debugging*** and it is often a very time-consuming “chore” when it comes to being a programmer. If you take your time to carefully follow problem solving steps 1 through 3, this should greatly reduce the amount of bugs in your programs and it should make debugging much easier.

STEP 6: Evaluate the Solution:

Once your program produces a result that seems correct, you need to re-consider the original problem and make sure that the answer is formatted into a proper solution to the problem. It is often the case that you realize that your program solution does not solve the problem the way that you wanted it to. You may realize that more steps are involved.

For example, if the result of your program is a long list of numbers, but your intent was to determine a pattern in the numbers or to identify some feature from the data, then simply producing a list of numbers may not suffice. There may be a need to display the information in a way that helps you visualize or interpret the results with respect to the problem. Perhaps a chart or graph is needed.

It is also possible that when you examine your results, you realize that you need additional data to fully solve the problem. Or, perhaps you need to adjust the results to solve the problem more efficiently (e.g., your game is too slow).

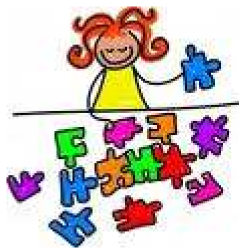
It is important to remember that the computer will only do what you told it to do. It is up to you to interpret the results in a meaningful way and determine whether or not it solves the original problem. It may be necessary to re-do some of the steps again, perhaps going as far back as step 1 again, if data was missing.

So there you have it. Those are the 6 steps that you should follow in order to solve problems using computers. Throughout the course, you should try to use this approach for all of your assignments. It is a good idea to practice problem solving to make sure that you understand the process. Below are some practice exercises that will help you practice the first 3 steps of the problem solving process. Later, you will gain experience with steps 4 through 6.

PRACTICE EXERCISES

Formulate a model and then develop an algorithm for each of the following problems. In each case, start with a simple algorithm and then try to think about situations that can realistically go wrong and make appropriate adjustments to the algorithm. Keep in mind that there is no "right" answer to these problems. Everyone will have a unique solution.

a. **Making a peanut butter and jam sandwich**



b. **Putting together a jigsaw puzzle**

c. **Playing the game of musical chairs**

d. **Replacing a flat tire on your car**

e. **Getting home from school today**

f. **Emptying a case of drinks into your refrigerator**

g. **Shopping for groceries (from entering store to leaving store)**

1.4 Control Abstraction

We just discussed the basics of producing an algorithm to solve a problem. However, it is not clear as to how much detail should go into our algorithm. For example, consider that you are at home on the couch and you are thirsty. How do you solve the problem ? Here is a simple algorithmic solution to your problem:

```
1.    go to kitchen
2.    open refrigerator
3.    choose a drink
4.    drink it
```

However, we could have come up with a more abstract (i.e., less detailed) algorithm as follows:

```
1.    get a drink
2.    drink it.
```

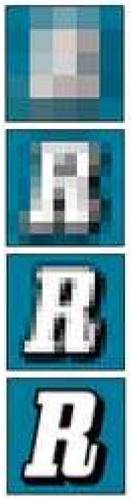
Or, we could have been much more detailed as follows:

```
1.     get off couch
2.     walk to kitchen
3.     open refrigerator
4.     if there is a carton of lemonade or orange juice then {
5.         take the carton
6.         close refrigerator
7.         go to the cupboard
8.         open cupboard
9.         take a glass
10.        close cupboard
11.        pour lemonade or juice into glass
12.        go to refrigerator
13.        open refrigerator
14.        put carton in refrigerator
15.        close refrigerator
       }
16.    otherwise if there is a soda then {
17.        take soda
18.        close refrigerator
19.        open soda
       }
20.    drink it
```

Each of these algorithmic solutions solves the problem. So then, which one is best ? The answer is not always easy. As a rule of thumb, we want to produce the simplest possible algorithm that is easily understandable. While it is important to provide enough detail to be able to properly describe the problem solution, it is also important not to get hung up on too much detail so that the algorithm becomes cluttered and overly complicated.

So, likely, the first of the three algorithmic solutions here would suffice as an adequate solution to the problem. This idea of coming up with a clear algorithm without too much details is known as abstraction.

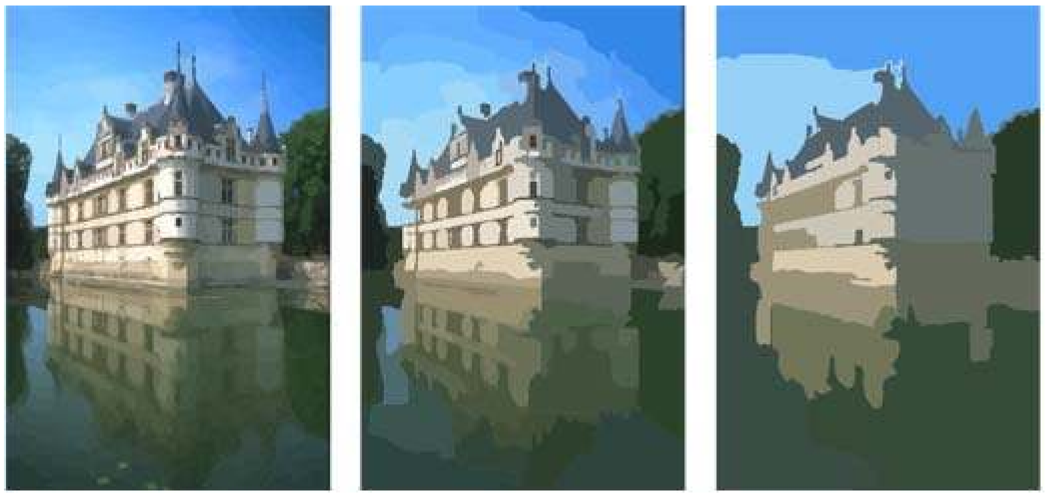
> Abstraction *is the process of reducing or factoring out details that are not necessary in order to describe an algorithm.*



Abstraction is important because it allows us to focus on a few concepts at a time. This allows us to get the "big picture" first in regards to the problem solution. We can then "fill in" the specific details at a later point in time. For example, in the above example, a statement like "**get a drink from the refrigerator**" would suffice when that step is part of a larger problem such as an algorithm that describes a person's daily routine around dinner time. However, if we needed to program such a step into a robot, then much more detail would be needed because the robot would require a more precise set of movements in order to carry out such an operation.

The analogy of image resolution can be used to describe abstraction. A low-resolution image has less detail and is more abstract than a high-resolution image. Abstraction allows us to lower the resolution of the image so as to save space (i.e., by hiding details) unless they are absolutely necessary.

We also see the idea of abstraction in 3D video game engines which hide details of objects further away as they are not necessary until the game character moves closer to those objects. It helps to speed up the game and reduce clutter while allowing the game player to focus on the more important objects nearby:



→ →→ → →→ → →→ → → **Abstraction** → → →→ → →→ → →→ →

We have just been discussing a kind of abstraction that allows our algorithm steps to be either quite abstract (i.e., high level) or more detailed (i.e., low level). This kind of abstraction is known as ***control abstraction***.

Often, in the cases where details are necessary, it is still possible to provide a higher-level algorithm, while allowing the more specific details to be described in a ***sub-algorithm*** or ***sub-program***. As a result, we are able to describe an algorithm at multiple layers of abstraction. For example, in our thirst-quenching scenario, we could use the details of our third very-specific algorithmic solution but hide the more detailed portions in a sub-algorithm as follows:

```
AlgorithmX: QuenchThirst
1.      get off couch
2.      walk to kitchen
3.      open refrigerator
4.      perform SubAlgorithm1
5.      close refrigerator
6.      drink it

SubAlgorithm1: GetDrink
1.      if there is a carton of lemonade or orange juice then {
2.          take the carton
3.          close refrigerator
4.          go to the cupboard
5.          open cupboard
6.          take a glass
7.          close cupboard
8.          pour lemonade or juice into glass
9.          go to refrigerator
10.         open refrigerator
11.         put carton in refrigerator
        }
12.     otherwise if there is a soda then {
13.         take soda
14.         open soda
        }
```

Notice how the main algorithm is quite simple now with much less detail. **SubAlgorithm1** is not a ***stand-alone algorithm*** in that it cannot solve the problem on its own.

For example, it assumes that the person is standing in front of the refrigerator and that the refrigerator door is open. We could also lessen these restrictions, for example, by incorporating steps **3** and **5** of the main **AlgorithmX** into **SubAlgorithm1** and moving them out of the main algorithm. Notice the changes:

```
AlgorithmX: QuenchThirst
1.        get off couch
2.        walk to kitchen
3.        perform SubAlgorithm1
4.        drink it

SubAlgorithm1: GetDrink
1.        open refrigerator
2.        if there is a carton of lemonade or orange juice then {
3.              take the carton
4.              close refrigerator
5.              go to the cupboard
6.              open cupboard
7.              take a glass
8.              close cupboard
9.              pour lemonade or juice into glass
10.             go to refrigerator
11.             open refrigerator
12.             put carton in refrigerator
          }
13.       otherwise if there is a soda then {
14.             take soda
15.             open soda
          }
16.       close refrigerator
```

Now we have abstracted out a little further by hiding some details within **SubAlgorithm1**. We could also perform additional layers of abstraction by abstracting within **SubAlgorithm1**:

```
AlgorithmX: QuenchThirst
1.        get off couch
2.        walk to kitchen
3.        perform SubAlgorithm1
4.        drink it

SubAlgorithm1: GetDrink
1.        open refrigerator
2.        if there is a carton of lemonade or orange juice then {
3.              perform SubAlgorithm2
          }
4.        otherwise if there is a soda then {
5.              take soda
6.              open soda
          }
7.        close refrigerator
```

```
SubAlgorithm2: PourDrink
1.      take the carton
2.      close refrigerator
3.      go to the cupboard
4.      open cupboard
5.      take a glass
6.      close cupboard
7.      pour lemonade or juice into glass
8.      go to refrigerator
9.      open refrigerator
10.     put carton in refrigerator
```

Notice how **SubAlgorithm1** became more abstract and easier to read. We can abstract out in this manner as often as we feel it to be necessary. When do we decide to stop this kind of abstraction ? Well, there is no fixed rule. However, when a sub-algorithm seems to be small enough to understand or when it relates to a well-defined real-life group of actions, then it is probably a good idea not to break it down any further.

However, it is possible that there is a portion of the algorithm that may be used by other algorithms. For example, what if we wanted an algorithm to place a glass on the kitchen table as part of getting prepared for dinner guests ? In that situation, you may realize that you will need to perform the same steps again as listed in lines **3 through 6** of **SubAlgorithm2**. Hence, it may be a good idea to create a **SubAlgorithm3** as follows:

```
SubAlgorithm3: GetGlass
1.      go to the cupboard
2.      open cupboard
3.      take a glass
4.      close cupboard
```

Then of course we could make use of this **SubAlgorithm3** within **SubAlgorithm2** as well as within our new algorithm for setting the table as follows:

```
SubAlgorithm2: PourDrink
1.      take the carton
2.      close refrigerator
3.      perform SubAlgorithm3
4.      pour lemonade or juice into glass
5.      go to refrigerator
6.      open refrigerator
7.      put carton in refrigerator
```

```
AlgorithmY: SetTableFor4
1.      walk to kitchen
2.      repeat 4 times {
3.          perform SubAlgorithm3
4.          place glass on table
5.          perform SubAlgorithm4    // similar algorithm to get plate from cupboard
6.          place plate on table
7.          perform SubAlgorithm5    // similar algorithm to get fork & knife from drawer
8.          place knife and fork on table
        }
9.      go back onto couch
```

Hopefully you see now how practical and powerful abstraction can be in making algorithms simpler, more readable and ultimately more understandable.

In computer science we give a special name to the sub-algorithms. They are sometimes called ***modules***, ***functions*** or ***procedures***. In fact, it is not a good idea to simply number all the sub-algorithms but instead to give them meaningful names. As standard convention, when naming a function or procedure, you should use letters, numbers and underscore (i.e., _) characters but not any spaces or punctuation. Also, the first character in the name should be a lower case letter. If multiple words are used as the name, each word except the first should be capitalized. Lastly, we often use parentheses (i.e., **()**) after the function or procedure name to identify it as a sub-algorithm. We'll see why later.

You should choose meaningful names that are not too long. Here are some meaningful names that we could use for our sub-algorithms:

SubAlgorithm1:	**chooseDrink()** or **chooseDrinkFromRefrigerator()**
SubAlgorithm2:	**pourDrink()** or **pourCartonDrink()**
SubAlgorithm3:	**getGlass()** or **getGlassFromCupboard()**
SubAlgorithm4:	**getPlate()** or **getPlateFromCupboard()**
SubAlgorithm5:	**getUtensils()** or **getForkAndKnife()** or **getUtensilsFromDrawer()**

Notice how readable the algorithms then become:

```
AlgorithmX: QuenchThirst
1.      get off couch
2.      walk to kitchen
3.      chooseDrink()
4.      drink it
```

and ..

```
AlgorithmY: SetTableFor4
1.    walk to kitchen
2.    repeat 4 times {
3.        getGlass()
4.        place glass on table
5.        getPlate()
6.        place plate on table
7.        getUtensils()
8.        place knife and fork on table
      }
9.    go back onto couch
```

Notice in **AlgorithmY** that we are getting a glass, a plate and a pair of utensils in lines **3**, **5** and **7**, respectively. Then on lines **4**, **6** and **8** we are placing these items on the table. Also, in line **3** of **AlgorithmX**, when we perform **chooseDrink()**, we are then drinking that drink on line **4**.

Hence each of these sub-algorithms go off and come back (i.e., return) with some kind of object (i.e., drink, glass, plate or utensils). When a sub-algorithm comes back with some kind of object (such as the glass or plate) or value (such as a numerical result), we call the sub-algorithm a ***function***.

If the sub-algorithm does not return any particular value, it is instead known as a ***procedure***. An example of a procedure would be a sub-algorithm for walking to the kitchen (perhaps called **walkToKitchen()**) and could be used on line **2** of **AlgorithmX** or line **1** of **AlgorithmY**. It is clear that such a sub-algorithm is not a function because although it does perform some actions, it does not come back with some kind of resulting object or value. It simply gets the person (or robot) to the kitchen and has no value to return.

When solving problems, it is often necessary to break the problem down into manageable steps as shown above. We do the same thing in real life so as to simplify the problem-solving process. For example, consider doing a jigsaw puzzle. If the jigsaw puzzle has many pieces (e.g., 1000), it can be an overwhelming problem to put it together. Some do not even enjoy such a task, yet others enjoy the challenge because they break it down into simpler, more easily-managed sub-problems.

A typical person may solve the puzzle as follows:

Algorithm: SolvePuzzle

```
1.      pour out all pieces on the table
2.      flip all pieces over to show the picture side
3.      separate the edge pieces from the inside pieces
4.      solve edge pieces
5.      repeat until there are no more easily identifiable pieces {
6.          select pieces that form an easily identifiable part of the picture
7.          put these selected pieces together
8.          try to fit picture portion to border or to other picture portions
        }
9.      repeat until no more pieces remain {
10.         pick piece up and try to fit it somewhere
11.         if its location is unclear, put piece back in pile
        }
```

Notice how the seemingly difficult problem is broken down into well-defined stages (or steps). By solving the edge pieces first, it gives the person a better idea as to the size of the whole image and it gives a feel for how the other parts of the puzzle will fit together.

In fact, steps 5 through 8 can be made into more specific steps if we had more information about the puzzle. For example, consider how you would solve this image:

GOLDEN
PUPPIES

Perhaps this would be your solution:

```
Algorithm: SolvePuppies
1.    pour out all pieces on the table
2.    flip all pieces over to show the picture side
3.    separate the edge pieces from the inside pieces
4.    solve edge pieces
5.    solve the puppies
6.    solve the sign
7.    solve the grass part
8.    solve the gate
9.    solve the hay
10.   solve the barn
```

Notice how specific we can be now with our algorithm because we know the exact picture that we are trying to build. Of course, the order in which we solve the particular parts of the image is unimportant, but do you see how breaking a problem down into simpler, smaller procedures can make it easy ?

Programming computers should be done in the same way. Always break your problem down into simple pieces that you understand. Keep breaking it down until you have a simple problem that you can understand.

This strategy of breaking down the problem into smaller pieces is often called ***divide and conquer*** and it represents the fundamental principle for problem solving.

It is not a difficult strategy. As mentioned, we do this every day naturally in real life. Even children can do this.

1.5 Algorithm Efficiency

Consider an algorithm for drawing a house. Since the problems is a little vague, there are many potential solutions. Here is one of them:

```
Algorithm1: DrawSimpleHouse
1.    draw a square frame
2.    draw a triangular roof
3.    draw a door
```

Obviously, we could have made a more elaborate house, but the solution above solves the original problem. What if this was our solution:

```
Algorithm2: DrawMoreComplexHouse
1.      draw a square frame
2.      draw a triangular roof
3.      draw a door
4.      draw windows
5.      draw chimney
6.      draw smoke
7.      draw land
8.      draw path to door
```

Which is a "**better**" solution ? That's not an easy question to answer. It depends on what "better" means. If time is of the essence (e.g., as in playing a game of Pictionary) then **Algorithm1** would be better because it can be drawn faster. The more elaborate house of **Algorithm2** would perhaps be "better" if visual appearance was the aim, as opposed to speed of drawing. This example brings up an important topic in computer science called *algorithm efficiency*.

Algorithm efficiency *is used to describe properties of an algorithm relating to how much of various types of resources it consumes. (Wikipedia)*

Normally in computer science we are interested in algorithms that are **time** and **space** efficient, although there are also other ways (i.e., metrics) for measuring efficiency. For example, **Algorithm1** is more efficient in terms of **time** but it is also more efficient in terms of **ink or pencil usage** as well as the amount of **space that it takes** on the paper. **Algorithm2** may be more efficient in terms of **detailing** in that, depending on the context, it may take longer for a person to guess what the drawing is (i.e., it could be confused with a barn, shed or dog house if this was drawn in a farm setting). In this case, the extra time taken to distinguish the house through the drawing of the windows and chimney may result in a quicker guess.

The ***runtime complexity*** *(a.k.a.* ***running time****) of an algorithm is the amount of time that it takes to complete once it has begun.*

The ***space complexity*** *of an algorithm is the amount of storage space that it requires while running from start to completion.*

Recall the following algorithm for setting a table:

```
AlgorithmY1: SetTableFor4
1.    walk to kitchen
2.    repeat 4 times {
3.        getGlass()
4.        place glass on table
5.        getPlate()
6.        place plate on table
7.        getUtensils()
8.        place knife and fork on table
      }
9.    go back onto couch
```

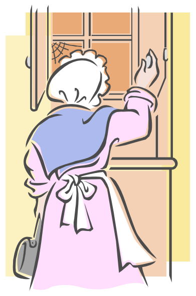
Why is this not an efficient real-world solution ?

It is inefficient in that it requires a lot of unnecessary travelling back and forth to the cupboard and table because it gets one glass at a time.

While this may be a safer solution for a small child, an adult would likely grab all 4 glasses at once as well as the plates and utensils.

Here is a different (yet similar) algorithm:

```
AlgorithmY2: EfficientSetTableFor4
1.    walk to kitchen
2.    getGlasses()
3.    place glasses on table
4.    getPlates()
5.    place plates on table
6.    getUtensils()
7.    place knives and forks on table
8.    go back onto couch
```



Notice that there is no longer a need for a "repeat loop" since we are getting all the glasses at once, all the plates at once and all the utensils at once.

We can actually generalize the algorithm to set the table for as many guests as we want by supplying some additional information in our functions.

A ***parameter*** *is a piece of data provided as input to a function or procedure*.

We can supply an arbitrary number in our algorithm to specify how many place settings to set as follows:

```
AlgorithmY3: EfficientSetTableFor8
1.    walk to kitchen
2.    getGlasses(8)
3.    place glasses on table
4.    getPlates(8)
5.    place plates on table
6.    getUtensils(8)
7.    place knives and forks on table
8.    go back onto couch
```

Notice that we supplied a number **8** between the parentheses of our function. This is where we normally supply additional information (i.e., parameters) to our functions. Now our function is clear as to how many place settings will be made, whereas **AlgorithmY2** was not clear.

But what would the **getGlasses()** function now look like ? Here was the 1-glass version:

```
GetGlass():
1.    go to the cupboard
2.    open cupboard
3.    take a glass
4.    close cupboard
```

Now we need to specify the parameter for the function and use it within the function itself:

```
GetGlasses(n):
1.    go to the cupboard
2.    open cupboard
3.    repeat n times {
4.        take a glass
      }
5.    close cupboard
```

Notice how the parameter is now being used within the function to get the necessary glasses. The value of **n** will vary according to how we call the function. For example, if we use **getGlasses(8)**, then within the function, **n** will have the value of **8**.
If we use **getGlasses(4)**, then within the function, **n** will have the value of **4**. So, the value for parameter **n** will always be the number that was passed in when the function was called.

For algorithms that are the most general, we often use the letter **n** as a kind of "placeholder" or "label" to indicate that we want the algorithm to work for any number from **0** to **n**. The "n" itself is not a special letter, it is just commonly used. So, the statement **getGlasses(n)** is indicating "get **n** glasses", where **n** may be any integer number that we want.

Obviously, there is a limit as to how many glasses a person could carry. However, to describe an algorithm in a very general way, we use **n** to indicate that our algorithm will work for any number from **0** to **n**. Using **n** instead of a fixed number, also allows us to compare two algorithms in regards to their efficiency. That is, we can often compare the number of steps that one algorithm requires with another algorithm.

For example, consider these two algorithms for setting the table for **n** people:

Algorithm A:

```
1.    walk to kitchen
2.    repeat n times {
3.          getGlass()
4.          place glass on table
5.          getPlate()
6.          place plate on table
7.          getUtensils()
8.          place knife and fork on table
      }
9.    go back onto couch
```

Algorithm B:

```
1.    walk to kitchen
2.    getGlasses(n)
3.    place glasses on table
4.    getPlates(n)
5.    place plates on table
6.    getUtensils(n)
7.    place knives and forks on table
8.    go back onto couch
```

What if we defined “efficiency” in this example to refer to the “***number of times we walked back and forth between the cupboard and the table***” ? Which algorithm is more efficient ? Well each time through the loop, **AlgorithmA** makes **3** trips between the cupboard and table. Since there are **n** place settings (i.e., **n** times through the loop), then the whole algorithm takes **n** x **3**, or **3n**, steps. What about **AlgorithmB** ? It takes only **3** trips between the cupboard and table altogether, regardless of how many place settings will be required.

So what can we conclude ? If we are setting a place for **1** person, either algorithm is good. If setting for **2** people, then **Algorithm B** is ***twice*** more efficient than **Algorithm A** since it requires ***half*** the travel between the cupboard and table. As **n** gets larger, the difference becomes more significant. For example, if we are setting the table for **8** people, then **Algorithm A** uses **8 times** (total of **24**) more trips than **Algorithm B** (which takes 3 trips).

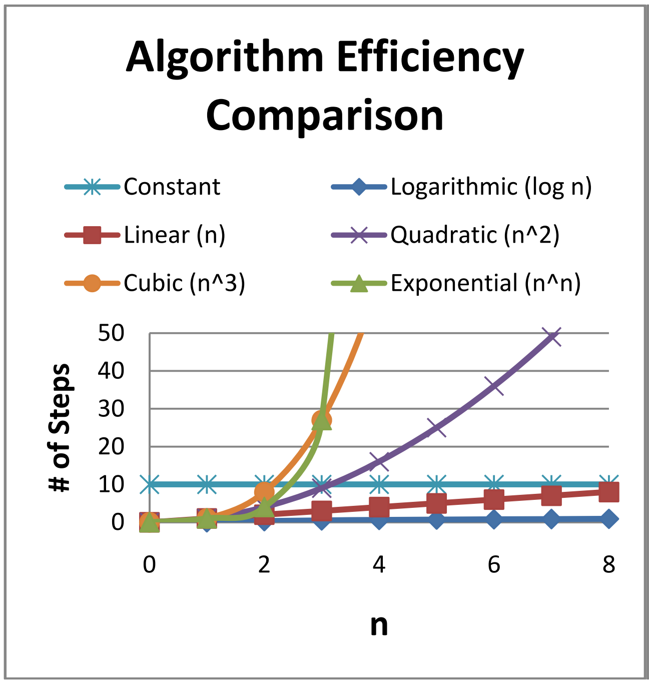
Regardless of the number of place settings, **Algorithm B** has a fixed cost of **3** (in regards to back and forth travels). Since this cost is fixed, we say that the algorithm has ***constant*** efficiency in terms of our particular cost metric.

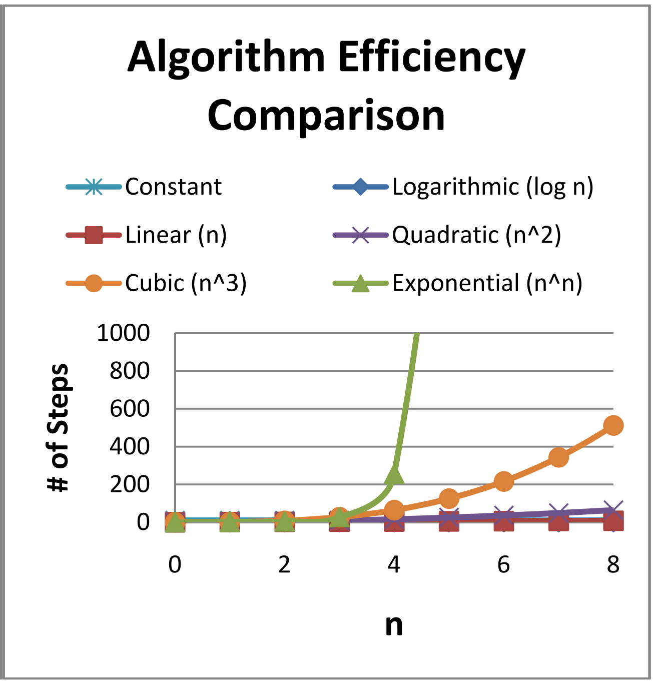
In contrast, **Algorithm A** is said to be ***linear*** in that the efficiency grows equally with respect to the value of **n**. Sometimes an algorithm has a constant value times **n** (e.g., **3n**). Since the **3** is constant (i.e., fixed in our case, because we have exactly **3** kinds of items that we are placing), the algorithm is still considered to be linear.

If we were to vary the **3** items to be **n** items (e.g., place 8 items at each of the 8 people's place settings, or 12 items at each of the 12 person's place settings), then we would end up with an **n** x **n** (or $\mathbf{n^2}$) algorithm which is called ***quadratic***.

Other common algorithm efficiency measures are ***logarithmic*** (i.e., **$\log_2 n$**), ***cubic*** (i.e., **n^3**) and ***exponential*** (i.e., **n^n**) … just to name a few.

Here are two graphs comparing various algorithm efficiencies as the value of **n** grows (graphs shown at two different scales):





Notice that the ***logarithmic***, ***constant*** and ***linear*** algorithms are insignificant when compared to the ***quadratic***, ***cubic*** and (especially) ***exponential*** algorithms. You may notice as well that the ***linear*** algorithm eventually passes the ***constant*** algorithm for larger values of **n**.

You may also notice that for very small values of **n**, the efficiency is generally not a big factor but that the efficiency can quickly become an issue for larger values. ***Exponential*** algorithms, for example, are usually unreasonable (i.e., useless) in practice except for very small values of **n**.

Logarithmic solutions are often preferred since they are significantly more efficient than even linear algorithms. For example, if **n** is **1,000,000** then a ***linear*** algorithm can take **1,000,000** steps whereas a ***logarithmic*** algorithm may take only **20** steps.

Sometimes it is hard to think in terms of an unknown number **n** because we are used to working with actual concrete numbers.

For example, assume that 16 players entered a one-on-one tennis tournament. If there can be no tie games how many games must be played if each player can be eliminated by one loss ?

We can figure this out with a simple table, doing a round-by-round count:

Round	Players Remaining	Games Played
1	**16** (nobody played yet)	**8**
2	**8** (winners from round 1)	**4**
3	**4** (winners from round 2)	**2**
4	**2** (winners from round 3)	**1**
		15 (total)

That was easy enough to figure out. But what if we had **n** players (assuming that **n** is an even number) ? What would the table look like ?

Round	Players Remaining	Games Played
1	**n**	**n/2**
2	**n/2**	**n/4**
3	**n/4**	**n/8**
4	**n/8**	**n/16**
5	**n/16**	**n/32**
…	**…**	**…**
…	**2**	**1**
		??? (total)

The answer seems to be (**n/2 + n/4 + n/8 + n/16 + n/32 + …**). But how far does that series of numbers go on ? We can try various values of **n** to find out. For example, if **n=100:**

(100/2 + 100/4 + 100/8 + 100/16 + 100/32 + …) = (50 + 25 + 12.5 + 6.25 + 3.125 + 1.5625 ...)

Hmmm…seems like the values approach 0 but never quite there. Try **n=128**:

(128/2 + 128/4 + 128/8 + 128/16 + 128/32 + …) = (64 + 32 + 16 + 8 + 4 + 2 + 1) = **127**

That one was easier. In fact, if you were to try various value of **n**, you would notice that the games placed would total **(n-1).** It works out evenly for games that are powers of 2 (e.g., n=2, 4, 8, 16, 32, 64, 128, 256, 510, 1024, etc). In fact, the number of rounds played is **$\log_2$(n)** (often written as just **log(n)** in computer science).

So, as you can see, trying to figure out efficiency with an unknown number is not always easy. Nevertheless, it is important for you as a computer scientist to understand how to write efficient algorithms that run faster, take up less computer memory or use less resources. We will briefly discuss efficiency at various times throughout this course. However, you will take further courses to investigate algorithmic efficiency much more thoroughly.

Chapter 2

Variables and Control Structures

What is in This Chapter ?

This chapter explains the notion of a ***variable*** which is a fundamental part of mathematics, problem solving and computer programming. Variables are used throughout a program with various ***control structures*** such as **if** statements as well as **for** and **while** loops. These are discussed with various examples. The idea of writing ***procedures*** and ***functions*** is explained in the context of the Processing language.

2.1 Problem-Solving With Variables

When we use a function in a program, it is necessary to store the result somehow so that it can be used later in the program. Remember a “standard” computer only evaluates one instruction at a time from your program. It is similar to the idea of having only one hand to perform an operation. For example, recall the example from chapter 1 where we needed to solve the thirst-quenching problem. If you wanted to perform **pourDrink()** with one hand, you would probably need to expand on the instructions by making use of the counter top to put things down occasionally so your hands are free:

Two-handed Algorithm

```
AlgorithmX:
1.    get off couch
2.    walk to kitchen
3.    go to refrigerator
4.    chooseDrink()
5.    drink it

chooseDrink():
1.    open refrigerator
2.    if there is a carton of juice then {
3.        pourDrink()
      }
4.    otherwise if there is a soda then {
5.        take soda
6.        open soda
      }
7.    close refrigerator

pourDrink():
1.    take the carton
2.    close refrigerator
3.    getGlass()
4.    pour lemonade or juice into glass
5.    go to refrigerator
6.    open refrigerator
7.    put carton in refrigerator

getGlass():
1.    go to the cupboard
2.    open cupboard
3.    take a glass
4.    close cupboard
```

One-handed Algorithm

```
chooseDrink():
1.    open refrigerator
2.    if there is a carton of juice then {
3.        pourDrink()
      }
4.    otherwise if there is a soda then {
5.        take soda
6.        open soda
7.        go to counter
8.        put soda down on counter
9.        go to refrigerator
      }
10.   close refrigerator
11.   go to counter
12.   pick up drink
pourDrink():
1.    take the carton
2.    go to counter
3.    put carton down on counter
4.    go to refrigerator
5.    close refrigerator
6.    getGlass()
7.    pick up carton
8.    pour lemonade or juice into glass
9.    put down carton
10.   go to refrigerator
11.   open refrigerator
12.   go to counter
13.   pick up carton
14.   go to refrigerator
15.   put carton in refrigerator
getGlass():
1.    go to the cupboard
2.    open cupboard
3.    take a glass
4.    put glass down on counter
5.    close cupboard
```

Notice all the changes that are necessary now because you have only one hand available. Since a typical computer program also has only one "hand" (i.e., processor) running your program, we will also have to make use of "counter tops" (i.e., ***storage space*** in the computer's memory) to "put down" (i.e., ***store***) the intermediate values/objects so that the program can continue performing other operations. The counter top is analogous to the computer's internal memory.

Notice in the algorithm that there were repeated trips back and forth from the counter to the refrigerator. This was because with only one hand, the glass and carton had to be put down in order for the refrigerator and cupboard doors to be opened and closed. When we go to the counter, that's like going to the computer's memory to store or retrieve something that we left there. Likewise, going back to the refrigerator is like going back to the task at hand (i.e., back from the computer's memory and ready to do something).

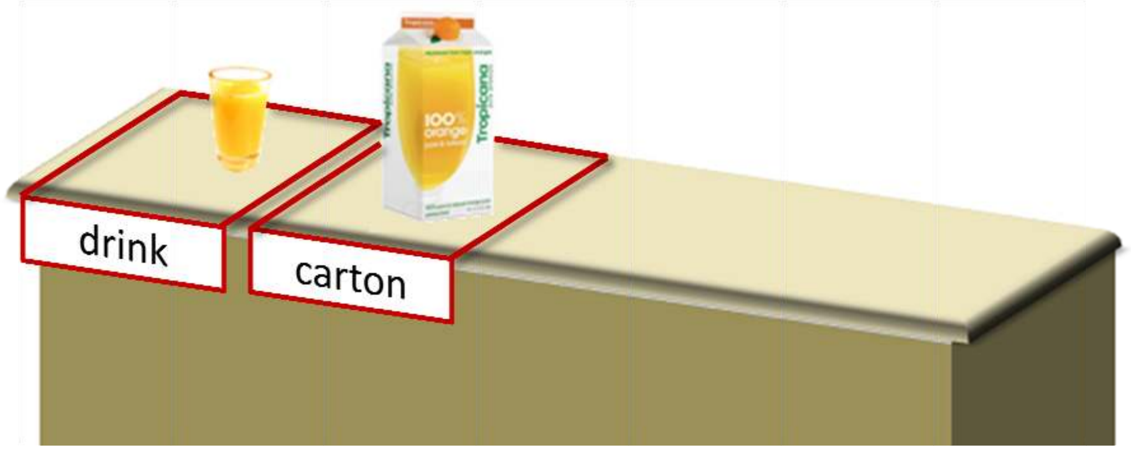
In reality, such details are usually an obvious part of the solution and need not be mentioned. However, it is important to indicate **what** information is being stored and **where** it is being stored so that we can use it later in the program. For example, what if we placed the carton of juice down somewhere but forgot where we put it ? We would not be able to fill up our glass later then. So, it is important in our algorithm to specify *when* and *where* we are storing information/data. A *variable* is used to store data:

A variable is a location in the computer's memory that stores a single piece of data.

A single algorithm or program may use many variables to store intermediate results. Each variable must be given a unique name so that it can be identified later.

Consider our one-handed algorithm for getting a drink from the refrigerator. Recall that we need to place both the **glass** (or soda) and possibly the **carton** on the counter top during the algorithm. In the **pourDrink()** function, for example, during steps **9** through **12**, both the **glass** and the **carton** are sitting on the counter. That should help us to see that we need two unique variables (i.e., two unique locations on the counter top) to store these objects.

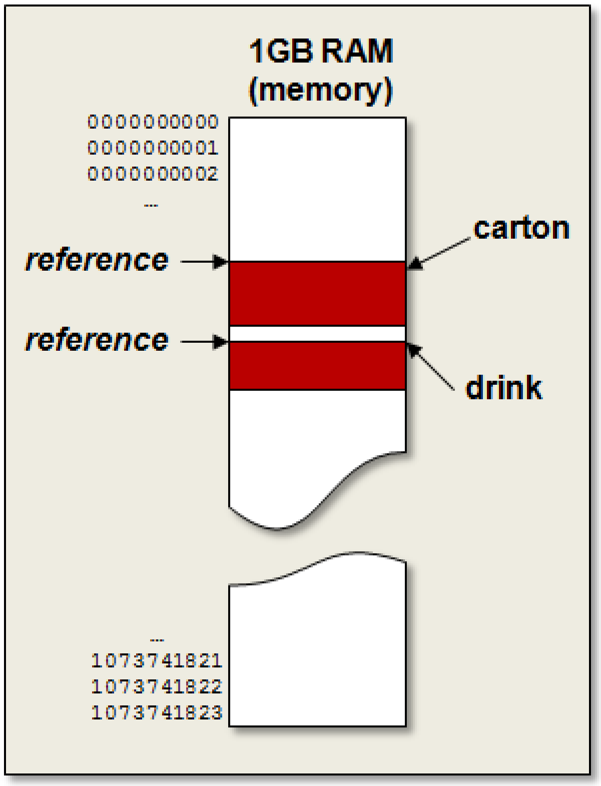
For each variable, we need to choose a meaningful ***variable name*** (i.e., a label) so that we can refer to it later. It makes sense to use the label "glass" to store the glass and "carton" to store the carton. However, when there is no juice, the algorithm gives back a soda can as the result. So, perhaps instead of "glass", we could use the label "drink" instead:



So, each variable that we make will ***have its own counter space on the counter top***. When we create (or ***declare***) a variable, we are really ***reserving space*** for an object on the counter top. And as with any reservation, we need to have a name for the reservation … which corresponds to the label (i.e., variable name).

When a variable's space has been reserved, usually there is nothing yet in that space … just the label. The term ***null*** is often used to indicate that *nothing* has been stored in the variable yet (i.e., noting is at that spot on the counter top). Once we put something in the variable (i.e., on the counter top at that label), the item that we place there is called the ***value*** of the variable.

Each variable that we declare (i.e., each time we reserve space for something), we are actually taking up space in the computer's memory. You may already know that your phone, your ipod, your flash drive, your computer etc… all have limited memory (or storage) space. The computer's memory space is called ***RAM***, which stands for (Random Access Memory). At the time that these notes were written, some phones had **512MB** (roughly 512 million bytes) of storage, while typical computers had **8GB** (roughly 8 billion bytes) of storage.



Consider, for example, a computer with 1GB of storage. Each time we declare a variable, we are reserving space in the RAM. That is, we are using up a portion of the computer's available memory. The bigger the object that we are storing, the more space that it takes up. So, here, we see that the carton would take up a little more space than a glass of orange juice.

The labels **carton** and **drink** are called ***references***, since they are used to ***refer to*** a particular object. The *reference* is actually just a number within the sequence of storage bytes in the RAM. It is also known as a ***memory address***, because it "sort of" represents the "home" of the object, as a real life address uniquely identifies a home in the real world.

As we will see later, there are simple kinds of objects (such as numbers and letters) called ***primitives*** that are stored in a simpler manner.

As with any real counter top, we can alter at any time what we place at that location on the counter. Similarly, we can change a variable's value at any time but putting a different value there. What happens to the old value ? It simply disappears.

In real life the object at a specific spot on the counter does not disappear when we put a new object there at the exact same location. So variables in a computer are a little different that real life. When it comes to replacing a variable's value with a new value, it is easiest to understand that process as ***over-writing*** the variable's value.

You may already have experience with over-writing information, perhaps from erasing mp3 songs from your ipod or phone, by putting new ones on the ipod/phone … the old songs are replaced (or overwritten) by the new ones. Variables are overwritten in the same manner.

So now, we should adjust our code to make use of the variables.

Notice what the code now looks like with two variables **drink** and **carton**:

```
AlgorithmX:
1.      get off couch
2.      walk to kitchen
3.      go to refrigerator
4.      drink ← result of chooseDrink()
5.      drink the drink

chooseDrink():
1.      open refrigerator
2.      if there is a carton of lemonade or orange juice then {
3.              drink ← result of pourDrink()
        }
4.      otherwise if there is a soda then {
5.              drink ← the soda
6.              open drink
        }
7.      close refrigerator
8.      return drink

pourDrink():
1.      carton ← the carton
2.      drink ← result of getGlass()
3.      pour carton contents into drink
4.      go to refrigerator
5.      put carton in refrigerator
6.      return drink

getGlass():
1.      go to the cupboard
2.      open cupboard
3.      drink ← the glass
4.      close cupboard
5.      return drink
```

Notice that the **drink** variable represents either a can of soda, an empty glass or a glass with lemonade or orange juice in it, depending on the line of the program. The ← is used to indicate that something is to be stored in the variable (i.e., that we want to give the variable a new value).

Notice as well how **go to the counter** and **put soda down on counter** and **go to refrigerator** are all combined into one storage step as **drink** ← soda. That's because leaving the refrigerator and heading over to the counter was done as a means to store the soda can on the countertop so that the single hand is free again to close the refrigerator. The entire storage process is now specified with just one instruction.

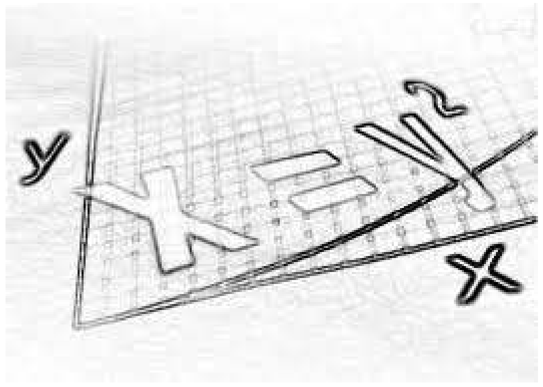
Similarly, the **go to counter** and **pick up drink** combination as well as the **go to counter** and **pick up carton** combination is analogous to simply getting the value of the variable in that it gets the object stored at that location on the countertop. The functions **chooseDrink()**, **pourDrink()** and **getGlass()** all bring back (i.e., ***return***) a drink, whether it is a full glass, an empty glass or a soda can. The **drink** is returned (from each function) back to the function that called it.

The return value is the value returned as a result of the function.

The idea of a *return value* becomes more obvious when the function is mathematical. For example, **sine(90)** is naturally understood to *return* the *value* **1** and **squareRoot(100)** would naturally *return* the *value* **10**.

One more point to mention regarding the variables is with respect to where (and how often) they are used. You will notice that the **drink** variable is used throughout the algorithm, whereas the **carton** variable is used only within the **pourDrink()** function. Variables that are used only with a single function/procedure (e.g., **carton**) are called ***local variables*** because they are only used locally (i.e., within the vicinity of the function or procedure). In contrast, variables that are used across many functions/procedures (e.g., **drink**) are known as ***global variables***.

Recall from chapter 1 that a *parameter* is a piece of data that is provided to a function. We used a parameter to represent the number of glasses/plates and utensils that we wanted to get from the cupboard: **getGlasses(8)**. We also used parameters to specify the values for drawing our houses such as the (**x**, **y**) coordinates and **width** X **height** dimensions.



A parameter is similar to a variable because it is a value that is used in your program. A parameter is different, however, in that it usually represents a value that remains constant within the context of where it is being used.

For example, when performing the **getGlasses(8)** function, the value of 8 is fixed (i.e., unchanging) while we are getting the glasses. Also, when we call **rect(100, 50, 100, 100)** in Processing to draw a rectangle, these 4 parameters are fixed/constant while the rectangle is being drawn. Since you (the algorithm designer and/or programmer) came up with these constant values, we can say that these values represent incoming ***algorithm parameters***.

In general, an algorithm may have many initial parameters and like variables they are usually given names. So inside an algorithm or computer program a parameter will usually look just like a variable. However, you should understand that these parameters will not change throughout the execution of the algorithm/program.

Later, you will create your own functions and procedures that may take incoming parameters. You will assign a name/label to these incoming values. You should understand though that the values of the parameters will NOT change throughout the function/procedure.

When developing your own algorithms to solve a problem, it is important to understand the difference between what has already been given to you (i.e., parameters) and what you need to figure out on your own. In the following examples, see if you can develop a computational model and identify the incoming parameters and variables that you will need.

Example:

Bob and Steve went on a vacation together. During the trip Bob paid for all the food and for the hotel. Steve paid for the gas and for the entertainment. Write an algorithm to compute the amount of money that Bob owes Steve (or Steve owes Bob) after the trip, assuming that they decided to split the expenses evenly ?

How many algorithm parameters are there ? What are they ?

```
Algorithm: TripExpenses
    f:    food cost
    h:    hotel cost
    g:    gas cost
    e:    entertainment cost

1. each ← (f + h + g + e) / 2
2. difference ← each – (g + e)
3. if difference < 0 then Bob owes Steve the difference
4. otherwise steve owes bob the difference
```

Example:

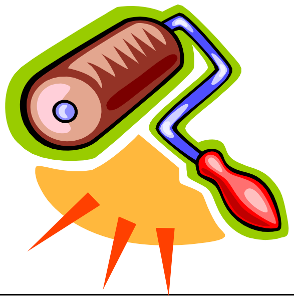
There are **n** kids in a room. As it turns out, some kids have socks on (with or without shoes), some kids are wearing shoes (with or without socks), and some kids are wearing both socks & shoes. Develop a computational model & algorithm to determine how many kids are barefoot?

```
Algorithm: Barefoot
    n:    # of kids in total
    x:    # kids with socks on
    y:    # kids with shoes on
    z:    # kids with socks & shoes on

1. socksOnly ← x – z
2. shoesOnly ← y – z
3. print (n – z – socksOnly – shoesOnly)
```

Can you re-write the algorithm without using the **socksOnly** and **shoesOnly** variables ?

Example:

A team of **n** people work *together* painting houses for the summer. For each house they paint they get **$256.00**. If the people work for **4** months of summer and their expenses are **$152.00** per month, how many houses must they paint for each of them to have one thousand dollars at the end of the summer?



```
Algorithm: PaintHouses
    n:    # people on team

1.  goal ← ((n * $1000) + (4 months * $152 per month))
2.  houses ← (goal / $256 per house)
3.  print houses
```

Notice that the **1000**, **4**, **152** and **256** here are all fixed/constant values, as opposed to parameters.

A constant is a single piece of data that does not change throughout the algorithm

In a more general version of this problem, we can make any (or all) of these values to be adjustable parameters.

In each of the above examples, the parameters and variables are all numbers. When programming, however, sometimes the variables and parameters may be of a different nature. For example, sometimes the input to an algorithm may be in the form of text, such as a person's name or an address. Or perhaps the variables come in the form of yes/no answers (i.e., true/false). Consider these examples:

Example:

Assume that you are given a name of a person in one of the following formats:

“Firstname Lastname”
“Lastname**,** Firstname”
“Firstname MiddleInitial**.** LastName”
“Lastname**,** Firstname MiddleInitial**.**”

You want to develop an algorithm that will determine whether or not the name is properly formatted to one of these formats. The output should be “YES” if properly formatted, otherwise “NO”. How would you write the algorithm ? Assume that the **name** is the only incoming parameter in the form of a bunch of consecutive characters.

```
Algorithm: NameFormat
    n:        the name

1.  if (n does not start with capital letter) then print NO
2.  if (n has a lower case letter after a space character) then print NO
3.  if (n has weird characters in it) then print NO
4.  if (n has a comma and there is no space after it or no letter before it) then print NO
5.  if (n has a middle initial that has more than one character) then print NO
6.  otherwise print YES
```

Example:

Assume that you want to take a vote among **5** friends to find out whether or not they agree to some issue (e.g., like not wearing speedos at a pool party). Each person votes yes or no. Develop an algorithm that determines the majority response (either yes or no).

To begin the algorithm, consider first getting all 5 votes and storing them. Then use the votes to determine the majority.

```
Algorithm: Majority
    v1, v2, v3, v4, v5:          the votes of the 5 people

1.   yesCount ← 0
2.   noCount ← 0
3.   if v1 = yes then yesCount ← yesCount + 1
4.   otherwise noCount ← noCount + 1
5.   if v2 = yes then yesCount ← yesCount + 1
6.   otherwise noCount ← noCount + 1
7.   if v3 = yes then yesCount ← yesCount + 1
8.   otherwise noCount ← noCount + 1
9.   if v4 = yes then yesCount ← yesCount + 1
10.  otherwise noCount ← noCount + 1
11.  if v5 = yes then yesCount ← yesCount + 1
12.  otherwise noCount ← noCount + 1
13.  if yesCount > noCount then display YES
14.  otherwise display NO
```

In this example, the yes and no parameters are considered to be **boolean** values.

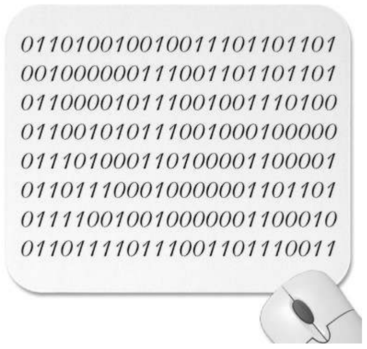
*A **boolean** is a value that is either **true** or **false**.*

So we can say that **yes** is the same as **true** and **no** is the same as **false**.

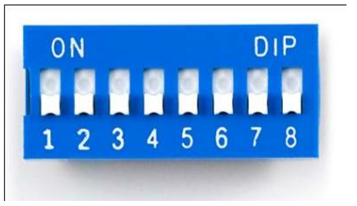
Now that you understand how to identify algorithmic parameters and variables, as well as how to develop and use simple computational models, it is important to discuss how these variables are represented in a real computer.

2.2 Variable Representation

All information in the computer is actually stored in the electronics as voltages … high and low voltages that can be thought of as billions of 1's and 0's that have some kind of meaning to them. That is, all user information (whether it is a name, phone number, picture, email, database, game, etc..) is stored as 1's and 0's which we call ***bits***.

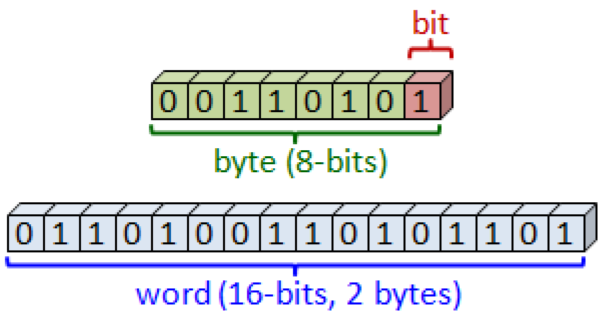


As humans, we have a hard time working at such a low level. We do better working with things like numbers, characters and real-world objects. So, rather than work with single bits, we group these bits into more abstract or higher-level packages.



A group of 8 bits is known as a ***byte***. A byte can represent 256 combinations of 1's and 0's. That is, if we think of each bit as being a switch which is either on or off, we can flip the 8 switches in 256 unique combinations. This allows a single byte to store a number from **0 to 255**.

When two bytes are required to represent a number, the pair of bytes is called a ***word***. A *word* can store a number in the range from **0 to 65535**. We can continue to group bytes together to store even larger integer numbers.



So, the *bit* and the *byte* are the two most primitive forms that a computer uses for number representation. Most computers will use the term ***boolean*** to represent a 0 or 1, but instead of saying "0" or "1" the terms "***false***" and "***true***" are used.

Bytes can also be used to represent letters, digits, punctuation, etc. which are called ***characters***. How so ? Well, back in 1968 it was decided that computers conform to a numbering standard called ***ASCII*** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange). That is, each combination of numbers in the range from 0 through 127 was mapped to (i.e., corresponds to) a particular keyboard character.

Here is the ASCII table (provided as a reference only … DO NOT try to memorize it):

ASCII value	Character(s)
0	null
1-31	various special characters
10	line feed
13	carriage return
32	space
33-47	!”#$%&'()*+,-./
48-57	0123456789
58-64	:;⇔?@
65-90	ABCDEFGHIJKLMNOPQRSTUVWXYZ
91-96	[\]^_`
97-122	abcdefghijklmnopqrstuvwxyz
123-127	{|}~□

So, for example, the letter "**A**" corresponds to number **65** in the ACSII table which is number **01000001** in binary bits (we will not discuss bit representation any further in this course). There are also versions of extended ACSII tables covering the numbers from 128 to 255. In addition, since computers began to be used internationally, 256 combinations were not enough to represent the letters of various international languages. Therefore, a new standard called ***Unicode*** has been developed (and continues to be expanded) to account for the other characters. However, a single byte is no longer sufficient to represent the character … two or more bytes are required.

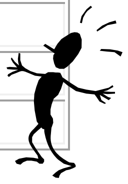
In addition, we can actually use bytes to represent real numbers (also called floating-point numbers) such as **3.14159265**. Also, by making assumptions on one particular bit in a byte (i.e., the ***most significant bit***, also called the ***sign bit***), we can allow the numbers to be either negative or positive. There are many details regarding number representation, but we will not discuss them further in this course.

The point is that *bits* are grouped to form *bytes* (or characters) which are also grouped to form larger numbers. Ultimately, this leads to what are known as ***primitive data types*** that are used in most programming languages. Here are the four basic primitives that are available in most programming languages (although the names may differ in each language):

- boolean – **true** or **false**
- integer – a positive or negative whole number
- floating-point number – a positive or negative real number with decimal places
- character – a letter, digit, punctuation or some other keyboard character

These are called primitive because they are the most **basic** types of data that we can store on the computer. Some languages will further distinguish between various types of integers or floats. For example, the following are the four official primitive data types in Processing (& JAVA) that can represent integers of various sizes:

Type	**Bytes Used**	**Can Store an Integer 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



Notice that the various types take up a different amount of memory space.

Similarly, there are two official primitive data types in Processing (& JAVA) to store floating-point numbers:

Type	**Bytes Used**	**Can Store a Real Number Within this Range**
float	4	-10^{38} **to** $+10^{38}$
double	8	-10^{308} **to** $+10^{308}$

Regardless of how we group the bytes, all information/data can be represented through the 4 basic primitives of boolean, integer, floating point numbers and characters.

So why are we talking about this ? Well, when programming, some languages (like Processing and JAVA) ***force you to specify the types*** for all of your variables. That means, for every variable that you create, you must indicate its type and its name.

In Processing and JAVA, for example, in order to use a variable to store a primitive kind of value (e.g., a boolean, integer, floating-point number or character), you must specify in your program the **type** followed by the **name** (must be unique) of the variable. Here are some examples:

```
boolean      hungry;
int          days;
byte         age;
short        years;
long         seconds;
char         gender;
float        amount;
double       weight;
```

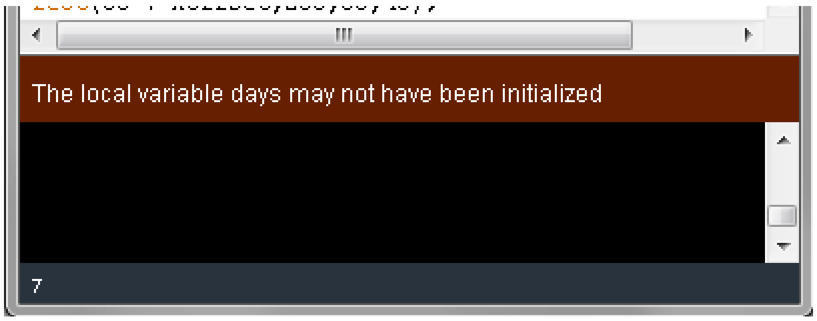
The above examples show all 8 primitive possible types that you may use in Processing/JAVA programs. Note that these will differ from language to language. Notice as well that there is a **;** character after each line, as with any step of an algorithm.

Each line above is responsible for ***declaring*** a variable. That means that a space is reserved in the computer's memory (with the given label) that can hold a value of the given type.

Declaring a variable, DOES NOT assign it any value, it only reserves space for the variable. In processing, after you declare a variable, you MUST give it a value ***before*** you use it. For example, suppose that tried to do this within one of your Processing functions:

```
int   days;
print(days);     // prints out the day variable's value
```

You would get the following error, preventing your program from running:



The local variable days may not have been initialized

7

Interestingly, Processing allows you to create **global** variables (i.e., variables outside of a function which are available through your entire program). In this case, it will assign a value of **0** to your variable and it will not produce an error.

A note about variable names … make sure to pick **meaningful** names that are not too long !! The name must be unique and it is case-sensitive (i.e., **Hello** and **hello** would not be considered the same).

Variable names may contain only letters, digits and the **'_'** character (i.e., no spaces in the name). As standard convention, multiple word names should have every word capitalized (except the first).

Here are some good examples of variable names:

- count
- average
- insuranceRate
- timeOfDay
- poundsPerSquareInch
- aString
- latestAccountNumber
- weightInKilograms

There is one more restriction when it comes to writing processing code. It is a good idea NOT to use **width** and **height** as variable names because these are variables that are already defined in Processing, which represent the width and height of the drawing area.

We use the term ***assign*** to represent the idea of "giving a value" to a variable. In Processing, the *assignment operator* is the **=** sign. So, we use **=** to put a value into a variable.

Here are a few example of how we can do this with some of the variables that we declared earlier:

```
hungry = true;
days = 15;
gender = 'M';
amount = 21.3f;  // (in JAVA only) floats must have an 'f' after them
weight = 165.23;
```

Something VERY important to remember when learning to program is that the value of the variable **must be the same type** of object (or primitive) **as the variable's type** that was specified when you declared it earlier. So for example, in the following table, make sure that you understand why the examples on the left are wrong, while the right examples are correct:

✗ `int days;` `days = 10.2789;`	✓ `int days;` `days = 10;`
✗ `boolean hungry;` `hungry = 'y';`	✓ `boolean hungry;` `hungry = false;`
✗ `char sex;` `sex = "F";`	✓ `char sex;` `sex = 'F';`

To help cut down the number of lines of code in our program, we are allowed to both *declare* and *assign* a value to our variables all on one line. So, from our earlier examples, we can do the following:

```
boolean    hungry = true;
int        days = 15;
char       gender = 'M';
float      amount = 21.3f;
double     weight = 165.23;
```

A variable may be ***declared*** only once in the program, but we may ***assign*** a value to it multiple times.

Can you determine the output of this piece of code:

```
int   days;

days = 43;
print(days);     // prints out 43
days = 15;
print(days);     // prints out 15
```

So, variables can be ***re-assigned*** a value, but cannot be ***declared*** again. Therefore, the following code will NOT compile:

```
int days = 365;
print(days);

int days = 7;    // cannot declare days again
print(days);
```

Here are some more pieces of code. Do you know what the output is ?

```
int x;
int y;
x = 34;
y = 23;
print (x + y);
```

Here is a similar example. Notice in Processing (and JAVA) that we are allowed to declare multiple variables of the same type on the same line, each separated by a **','**:

```
int x, y;
x = 34;
y = x;
print(x + y);
```

Here is another one:

```
int x, y, z;

x = 3*2*1;
y = x + x;
z = x;
print(z);
```

Note that even though we use **x** a few times, it does not change its value.

Here is one that is a little more interesting:

```
int    total;
float  average;

total = 12 + 25 + 36 + 15;
average = total / 4;
print("The average is ");
print(average);
```

Here is the output:

```
The average is 22.0
```

Notice that the **print()** function also allows you to display a fixed set of characters defined within double quotes. This fixed set of characters is called a ***String***.

Each time we call **print()** , the information will appear on the same line, so it is important to have the extra space character at the end of the string above, otherwise the result would be crowded close to the text like this:

```
The average is22.0
```

We can also combine the two print statements into one line as follows:

```
print("The average is " + average);
```

This code will append the **average** variable's value to the string by using the **'+'** operator.

A similar function called **println()** is available that will allow you to stop printing on one line and start another:

```
println("The average is ");
print(average);
```

will produce:

```
The average is
22.0
```

If you have a value that will remain ***constant*** throughout your program you can use the keyword ***final*** (implying that it has its final value and will not change again) before the variable's type. In this case, you must assign the value to the constant when it is declared:

```
final int      DAYS = 365;
final float    INTEREST_RATE = 4.923;
final double   PI = 3.1415965;
```

Normally, constants use uppercase letters with underscores (i.e., _) separating words.

2.3 Conditional Statements

We have already seen the need to make decisions in our program based on various input and certain calculations. Recall, for example, the **TripExpenses** algorithm:

```
Algorithm: TripExpenses

1.  each ← ((f + h + g + e) / 2)
2.  difference ← (each – (g + e))
3.  if difference < 0 then Bob owes Steve the difference
4.  otherwise steve owes bob the difference
```

Notice here that a decision had to be made as to whether Bob owed Steve the difference or vice-versa. The **if/then/otherwise** here is known as a ***conditional statement*** (often simply called an ***if statement***).

Example:

Often, the **otherwise** part can be left off of an **if** statement. For example, consider developing a simple computational model that computes a price for patrons who want to go to the theatre. Assume that there is a discount of 50% for women that are senior (i.e., 65 or older) or to girls who are 12 and under. For all other people, the discount should otherwise be 0%.

Develop an algorithm that displays the appropriate discount for a particular person buying the ticket:

```
Algorithm: TheatreDiscount
        p:      person buying theatre ticket

1.      discount ← 0
2.      gender ← gender of p
3.      age ← age of p
4.      if gender is female and age > 64 or gender is female and age < 13 then {
                discount ← 50
        }
5.      print discount
```

Notice that there was no need for an **otherwise** statement here because the discount was set to zero and a decision was only necessary to set it to 50 in the two particular cases.

You may notice that step 4 is a little ambiguous because of the “and”s and “or”s being used. That is, notice how the decision differs based on the placement of parentheses:

if (**gender** is female and **age** > 64 or **gender** is female and **age** < 13) **then …**
if (**gender** is female and **age** > 64) or (**gender** is female and **age** < 13) **then …**
if (**gender** is female and (**age** > 64 or **gender** is female) and **age** < 13) **then …**
if (**gender** is female and (**age** > 64 or **gender** is female and **age** < 13)) **then …**

Which is the correct understanding of the problem ? The 2nd one. When programming, it is important to be as clear as possible in your code. Therefore, try to be aware of the need for parentheses when the code seems complex.

As it turns out, "**and**", "**or**" and "**not**" are common operators in computer science, called ***logical operators***. They allow you to work with Boolean values (i.e., true/false values) to combine them in logical ways in order to achieve an overall Boolean result.

For example, **age>64** results in either **true** or **false** as does **age < 13**. Also, **gender is female** will also produce a **true** or **false** result. When we **and/or** these **true/false** values together, we end up with an overall **true/false** result that is used by the **if** statement to decide whether or not to evaluate the code within the body of the **if** statement.

Below is a "truth table" explaining the results of using any two boolean values, say **b_1** and **b_2**, in an **if** statement:

b_1	**b_2**	**if (b_1 and b_2)**	**if (b_1 or b_2)**	**if (not b_1)**	**if (b_1)**
false	false	false	false	**true**	false
false	**true**	false	**true**	**true**	false
true	false	false	**true**	false	**true**
true	**true**	**true**	**true**	false	**true**

Notice that the **and** results in **true** only when both Booleans are **true**, and **false** otherwise. Conversely, the **or** results in **false** only when both Booleans are **false**, and **true** otherwise. Also note that the **not** results in the opposite value of the Boolean. Of course, we can combine multiple **and/or/not** operators within the same **if** statement as in our example.

Example:

Consider writing an algorithm that takes the number grade of a student (i.e., from **0%** to **100%**) and outputs a letter grade (from **F** to **A+**). To do this, we need to first understand the computational model … that is, which letter grade corresponds to which number grades:

A = 80% - 100%
B = 70% - 79%
C = 60% - 69%
D = 50% - 59%
F = 0% - 49%

Algorithm: GradeToLetter

```
        grade:    the number grade of a student

1.      if (grade is between 80 and 100) then
2.          print "A"
3.      if (grade is between 70 and 79) then
4.          print "B"
5.      if (grade is between 60 and 69) then
6.          print "C"
7.      if (grade is between 50 and 59) then
8.          print "D"
9.      otherwise
10.         print "F"
```

Notice how each **if** statement checks to see whether the **grade** lies within a specific range. We can re-write this using **and** operators as follows:

Algorithm: GradeToLetter2

```
        grade:    the number grade of a student

1.      if (grade >= 80 and grade <= 100) then
2.          print "A"
3.      if (grade >= 70 and grade <=79) then
4.          print "B"
5.      if (grade >= 60 and grade <= 69) then
6.          print "C"
7.      if (grade >= 50 and grade <= 59) then
8.          print "D"
9.      otherwise
10.         print "F"
```

This is more realistic when programming because very few (if any at all) programming languages have a "**between**" kind of command that allows you to check values within a certain range.

There is a small issue with the above code in regards to efficiency. The algorithm is correct, but it is not efficient. Assume that the grade entered was **92%.** Step **1** and **2** will be evaluated. However, the algorithm will then continue with steps **3**, **5**, and **7** by checking those **if** statements … which will all evaluate to **false** anyway.

We can avoid this inefficiency by created ***nested if statements***. This means that we insert successive **if** statements into the **otherwise** part of the earlier **if** statements as follows:

```
Algorithm: GradeToLetter3
        grade:      the number grade of a student

1.      if (grade >= 80 and grade <= 100) then
2.              print "A"
3.      otherwise {
4.              if (grade >= 70 and grade <=79) then
5.                      print "B"
6.              otherwise {
7.                      if (grade >= 60 and grade <= 69) then
8.                              print "C"
9.                      otherwise {
10.                             if (grade >= 50 and grade <= 59) then
11.                                     print "D"
12.                             otherwise
13.                                     print "F"
                        }
                }
        }
```

Notice how the successive **if** statements are shown indented, as they lie within the **otherwise** part of the previous **if** statement. What happens if **92%** is entered ? Steps **1** and **2** are evaluated, but then since steps **4** through **13** are inside the **otherwise** part of step 1's **if** statement, they are not evaluated. This is more efficient.

Notice as well the **if** statement of step **4**. Since it is in the **otherwise** part of step **1**'s **if** statement, the **grade** must be less than or equal to **79**, since if not, the algorithm would have stopped on step **2**. So, we don't need to check whether or not **grade <= 79** in step **4**. Likewise, the upper bound of **69** and **59** need not be checked in steps **7** and **10**. Here is the better code:

```
Algorithm: GradeToLetter3
        grade:      the number grade of a student

1.      if (grade >= 80 and grade <= 100) then
2.              print "A"
3.      otherwise if (grade >= 70) then
4.              print "B"
5.      otherwise if (grade >= 60) then
6.              print "C"
7.      otherwise if (grade >= 50) then
8.              print "D"
9.      otherwise
10.             print "F"
```

Notice how the **otherwise** statements are also aligned nicely one after another. Since there is only one line to evaluate in each **if** statement, we do not need the braces **{ }.** In such a scenario it is often the case that we line up the statements as shown, since it is more intuitive to read.

Example:

Consider another example in which we are given an integer representing a **month** and we would like to determine the number of days in that month (we will assume that it is not a leap year). Here is the table of information that we need to know to begin:

Month	**Jan**	**Feb**	**Mar**	**Apr**	**May**	**Jun**	**Jul**	**Aug**	**Sep**	**Oct**	**Nov**	**Dec**
Days	31	28	31	30	31	30	31	31	30	31	30	31

Here is the algorithm:

```
Algorithm: DaysInMonth
        month:    a month from 1 to 12

1.      if (month is 1) then
2.            print 31
3.      otherwise if (month is 2) then
4.            print 28
5.      otherwise if (month is 3) then
6.            print 31
7.      otherwise if (month is 4) then
8.            print 30
9.      etc..
…
```

However, you can see that the combined **if** statements will be 24 lines long! Since there are only 3 values for the months (i.e., 31, 30 and 28), there should be a way to arrange it in a format like this …

```
if (...) then                // Jan, Mar, May, Jul, Aug, Oct, Dec
   print 31
otherwise if (...) then      // Apr, Jun, Sep, Nov
   print 30
otherwise                    // Feb only
   print 28
```

We would just need to group the months into the appropriate category and use **or** operators to decide which ones match the corresponding **if** statement:

```
Algorithm: DaysInMonth2
        month:    a month from 1 to 12

1.      if ((month is 1) or (month is 3) or (month is 5) or (month is 7) or
           (month is 8) or (month is 10) or (month is 12)) then
2.            print 31
3.      otherwise if (month is 4) or (month is 6) or (month is 9) or (month is 11) then
4.            print 30
5.      otherwise print 28
```

This seems much shorter. How can we shorten the algorithm description even further ?

We can re-arrange the code to check for the 28 day month first, and the 31 day months last:

```
Algorithm: DaysInMonth2
        month:    a month from 1 to 12

1.      if (month is 2) then
2.            print 28
3.      otherwise if (month is 4) or (month is 6) or (month is 9) or (month is 11) then
4.            print 30
5.      otherwise print 31
```

Wow. The code works the same way, but is much shorter, cleaner and nicer. It is often the case that we can re-arrange our algorithm steps in this manner like this in order to make it more readable and simpler to understand.

Example:

Here is a larger, more complex example. See if you can understand where you should use nested **if** statements.

Consider writing a program that will be placed at a kiosk in front of a bank to allow customers to determine whether or not they qualify for the bank's new "Entrepreneur Startup Loan". Assume that this kind of loan is only given out to someone who is currently employed and who is a recent University graduate, or someone who is employed, over 30 and has at least 10 years of full-time work experience.

The program should display information to the screen as well as ask the user various questions … and then determine if the person qualifies.

What questions should be asked ?

- Are you currently employed ?
- Did you graduate with a university degree in the past 6 months ?
- How old are you ?
- How many years have you been working at full time status ?

Here is an algorithm:

Algorithm: LoanQualificationKiosk

```
1.  print welcome message
2.  employed ← ask user if he/she is currently employed
3.  hasDegree ← ask user if he received a university degree within past 6 months
4.  age ← ask user for his/her age
5.  yearsWorked ← ask user for # years worked at full time status

6.  if (employed is true) then {
7.      if (hasDegree is true) then
8.          print "Congratulations, you qualify!"
9.      otherwise {
10.         if (age >= 30) then {
11.             if (yearsWorked >= 10) then
12.                 print "Congratulations, you qualify!"
13.             otherwise
14.                 print "Sorry, you do not qualify.   You must have
                        worked at least 10 years at full time status."
            }
15.         otherwise
16.             print "Sorry, you do not qualify.   You must be a
                      recent graduate or at last 30 years of age."
        }
    }
17. otherwise
18.     print "Sorry, you must be currently employed to qualify."
```

You may have noticed that some **if** statements are nested within others. Of course, the order that the **if** statements are evaluated in can vary. That is, the check in step 6 for employment can be done after the check for the degree, age and years worked. However, since employment is necessary in all special cases, it is good to check for that first so that the user code completes quicker when the user is unemployed.

In fact, we could intermix the user input (from lines 2 through 5) with the **if** statements as follows:

Algorithm: LoanQualificationKiosk

```
1.    print welcome message
2.    employed ← ask user if he/she is currently employed
3.    if (employed is false) then
4.        print "Sorry, you must be currently employed to qualify."
5.    otherwise {
6.        hasDegree ← ask user if he received a univ. degree within past 6 months
7.        if (hasDegree is true) then
8.            print "Congratulations, you qualify!"
9.        otherwise {
10.           age ← ask user for his/her age
11.           if (age < 30) then
12.               print "Sorry, you do not qualify.  You must be a
                         recent graduate or at last 30 years of age."
13.           otherwise {
14.               yearsWorked ← ask user for # years worked at full time status
15.               if (yearsWorked >= 10) then
16.                   print "Congratulations, you qualify!"
17.               otherwise
18.                   print "Sorry, you do not qualify.  You must have
                             worked at least 10 years at full time status."
              }
          }
      }
```

This code may seem a little more cluttered, but it has the advantage that the program ends quickly and abruptly as soon as any information is entered from the user that disqualifies him/her. After all, there is nothing more annoying that having to fill out a form with a lot of information in it only to find out that the first piece of information disqualified you!

In time, you will get used to adjusting your code accordingly to make it more efficient and user-friendly.

2.4 Counting

Consider a common situation in which you want to count the total of a set of values. For example, assume that you are having a pizza lunch and you want your employees to tell you how many slices they each want. Your goal is to determine how many pizzas to buy (assume 8 slices per pizza … and all just plain pepperoni). Write an algorithm for doing this, assuming that there are **n** people:

Algorithm: PizzaCount

```
1.      start with a total of 0 slices
2.      repeat n times {
3.              ask a person for the number of slices that they want
4.              add those slices to the total
        }
5.      divide the total slices by 8 and print the answer
```

It seems straight forward. Here is a more formal version:

Algorithm: PizzaCount

```
1.      total ← 0
2.      repeat n times {
3.              number ← ask a person for the number of slices that they want
4.              total ← total + number
        }
5.      print total / 8
```

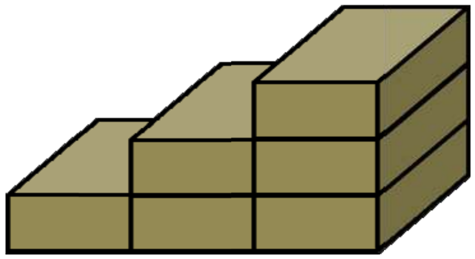
The code is logical and straight forward. In real life though, the code will usually produce a non-integer number (e.g., **4.3** pizzas). In this example we would want to increase **4.3** to the whole number **5** so that we have an integer number of pizzas. In math, the ***ceiling*** function is used to get the nearest integer *above* the given real value (likewise, the ***floor*** function gives us the nearest integer *below* the given value).

Example:

What would differ in the above code if we wanted to find the average number of pizza slices that each person will eat ? Only one thing will change… can you figure it out ?

Example:

Assume that we want to stack concrete slabs on top of each other to form a staircase. Develop an algorithm that will determine how many slabs would be needed to create a staircase **n** stairs high ?



To begin, you should realize that **n** may be a very large number. What is our mathematical model ? This is how many numbers we need:

1 + 2 + 3 + 4 + … + n

Some of you may realize that this value can be computed as **n(n+1)/2**. However, assume that we are unaware of that nifty formula. How would you go about solving this problem ? You might realize that some kind of counter is required (i.e., a variable) and that we need to keep adding an increasingly large integer to the count. Here is one possible solution:

Algorithm: Stairs

```
1.    total ← 0
2.    currentHeight ← 1
3.    repeat n times {
4.        total ← total + currentHeight
5.        currentHeight ← currentHeight + 1
      }
6.    print total
```

The above algorithm demonstrates a very popular form of repetition in computer programs … that of counting from **1** to some fixed value (i.e., **n** in this case). You may notice that the counter is called **currentHeight** and that it starts at 1 but each time through the loop it increases by 1. Hence, **currentHeight** goes through the values of 1, 2, 3, 4, …, **n**. These are exactly the values that we want to add together … and we do so with the **total** variable.

The **currentHeight** variable above is an example of a ***loop counter*** or ***loop index*** because it adds 1 each time through the loop. Most loops that you will use when you are programming will involve these simple forms of counters.

Example:

Another kind of counting involves searching through some items to enumerate (i.e., count) them. Perhaps we need to count the number of items that match some kind of search criteria. For example, how would we write an algorithm that went through a list of **n** people and count how many people were adults ?

Algorithm: CountAdults

```
1.   set the total of the adults to 0.
2.   repeat n times {
3.       get a person and look at his/her age
4.       if the person is 18 years of age or older then {
5.           add one adult to the total
         }
     }
6.   display the total number of adults
```

Can you identify the variables being used here ? You should have identified:

- **total** - since it changes during the program.
- **age** - since it is found in step **3** and used later in step **4**.

Here is the more formal version:

Algorithm: CountAdults

```
1.   total ← 0
2.   repeat n times {
3.       age ← the age of a person
4.       if age >= 18 then {
5.         total ← total + 1
         }
     }
6.   print total
```

Example:

Suppose that we wanted to print out the even numbers from **1** to **100**. How could we do this ? Do you know how to check whether or not a number is even ? We can check if the remainder after dividing by two is zero. The modulus operator (**%** in Processing and JAVA) gives the remainder after dividing. We just need to put this into a loop:

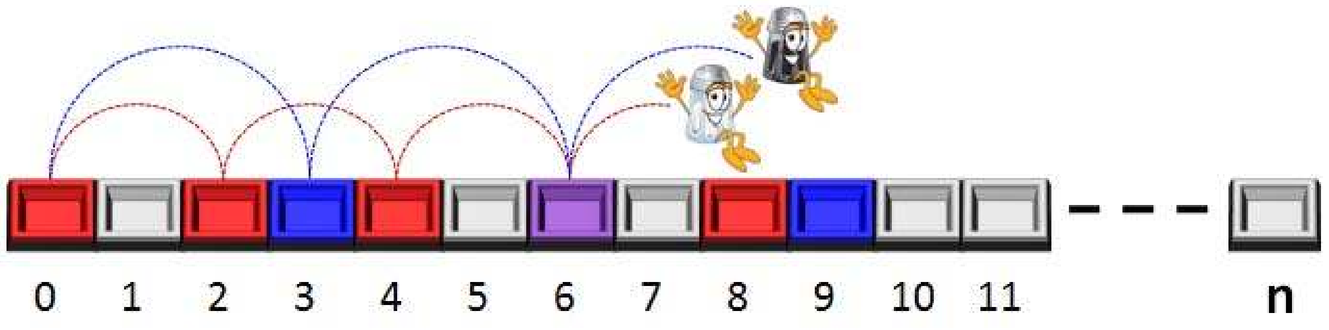
Algorithm: OddNumbers

```
1.      for each number n from 1 to 100 {
2.          if n modulus 2 = 0 then {
3.              print n
            }
        }
```

This kind of loop is called a ***for*** loop (or ***for each*** loop) because it repeats the loop for a set of particular values. Notice that it iterates through each location number from **1** through to **100** and that the numbers are used within the loop (i.e., in step 2). The number **n** itself is called the ***loop variable*** because it varies (i.e., changes) each time through the loop. In this case, the *loop variable* is the same as the *loop index* or *loop counter*, as it simply counts from **1** to **n**.

Example:

Imagine that you are creating a game where two players are moving along a one-dimensional grid (i.e., path). One player always jumps forward **2** steps at a time, while the other always jumps forward **3** steps at a time. Develop an algorithm that figures out how many grid locations have not been landed on if the two players start at the same location (i.e., 0) and they each jumped up to grid location **n**.



How do we approach this problem ? First, examine the grid locations covered by player 1:

0, 2, 4, 6, 8, 10, 12, 14, … (seems to be the even numbered locations)

and those covered by player 2:

0, 3, 6, 9, 12, 15, 18, 21, … (seems to be the locations that are multiples of 3)

We could try to figure out a formula… but can we do this with some kind of loop counter ? What if we examine each location at a time … can we determine by the location number whether or not it would be landed on ?

Algorithm: HopCoverage

```
1.     spotsNotLandedOn ← 0
2.     for each locationNumber from 0 to n {
3.         if the locationNumber is not divisible by 2 and not divisible by 3 then {
4.             spotsNotLandedOn ← (spotsNotLandedOn + 1)
           }
       }
5.     print spotsNotLandedOn
```

Again, we see the **for** loop used as a way of counting now from **0** to **n**.

Example:

How could we change the algorithm so that we displayed a list of all the locations that the two players met at along the way ?

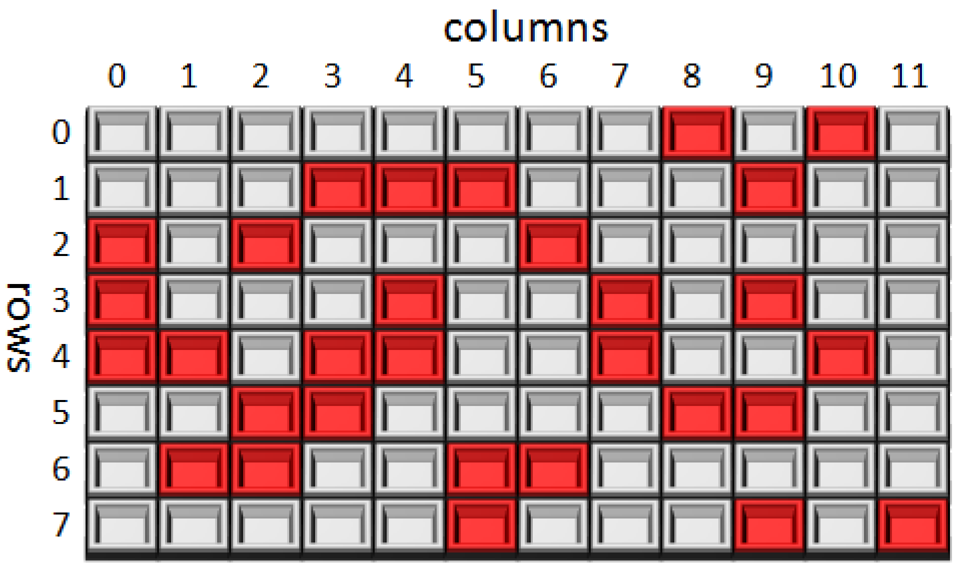
We would just need to find the locations that were multiples of 2 or 3:

Algorithm: HopMeetings

```
2.     for each locationNumber from 0 to n {
2.         if locationNumber is divisible by both 2 and 3 then {
3.             print locationNumber
           }
       }
```

Example:

Often, a situation arises where we have a 2-dimensional grid of locations. For example, imagine selling seats for an event taking place at a stadium. The seats are often arranged in rows and columns for each section of the stadium. Imagine that some seats are sold (red), but others are available (gray):



How could we write an algorithm that counted the available seats ?

Algorithm: CountSeats

```
1.      seatsAvailable ← 0
2.      for each seat {
3.          if the seat is available then {
4.              seatsAvailable ← seatsAvailable + 1
            }
        }
5.      print seatsAvailable
```

Its not too difficult to come up with this algorithm and it is logical. Now although the algorithm above is correct, step 2 is often not precise enough when it comes to programming. There would need to be a systematic way of getting each seat. Somehow, we need to indicate what order to get the seats in. Just think of how you would count the seats. You would likely count the available seats in each row and then go to the next row…and the next one… and so on until you completed all rows.

So, to do this, we would choose one row (e.g., row 0) and then loop through the seats in that row… which is the same as going through the columns of the grid. This loop would then need to be inside a bigger loop that made sure that we systematically went through each row. Here is what we would do:

Algorithm: CountSeatsSystematically

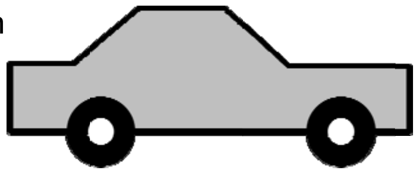
```
1.      seatsAvailable ← 0
2.      for each row {
3.          for each column {
4.              if the seat at this row and column is available then {
5.                  seatsAvailable ← seatsAvailable + 1
                }
            }
        }
6.      print seatsAvailable
```

Notice that the inner loop goes through the seats in a particular row and that this is repeated for each row. Whenever we include one loop inside another like this, it is known ad ***nested looping***, or simply ***nested loops***.

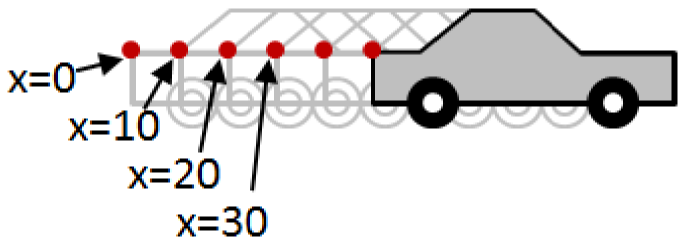
Nested loops are quite common in programming. They often appear in situations involving data arranged in a grid pattern such as applications that manipulate data tables, pictures or images, and graphics involving x/y coordinate systems.

Example:

Consider writing a program that will cause a car to be displayed on the window, moving across the window to the right. The car can be drawn easily, kinda like drawing the house as we did before.



Now we need to understand what is happening as the car moves. What changes as the car moves along to the right ? The horizontal position changes … which is the **x** coordinate of the car's points. So, we would need to introduce a variable, say **x**, to represent the horizontal location of the car and use it in our program. Likely the **x** refers to some fixed part of the car's image … perhaps the top/leftmost point:



Algorithm: DrawCar

```
1.      for successive x locations from 0 to windowWidth {
2.          draw the car at position x
3.          x ← x + 10
        }
```

windowWidth here is a fixed constant representing the width of the window. Notice how **x** starts at **0** and then continues to increase by some constant value (in this case 10). The value of **10** will represent the speed. If we use a smaller number, such as **5**, the car will appear to move slower across the screen, as it will move only half as far each time that we redraw it. A larger value, such as **20**, will double the speed.

A more "proper" way to write the **for** loop in our algorithm is to specify this fixed increase amount each time (called the ***loop increment***). We can re-write this as follows:

Algorithm: DrawCar2

```
1.      for x locations from 0 to windowWidth by 10 {
2.          draw the car at position x
        }
```

The "**by 10**" here is a way of saying that **x** will be increased by **10** each time the loop restarts.

Example:

Now what if we wanted the car to speed up ? The value of **10** would have to start smaller, perhaps at **1** and then increase. So the loop increment (i.e., the *speed* in this case) would need to increase each time as well as the **x** value.

Algorithm: AccelerateCar

```
1.      speed ← 0
2.      for x locations from 0 to windowWidth by speed {
3.          draw the car at position x
4.          speed ← speed + 2
        }
```

Notice the need for a new **speed** variable and how this variable is used to move the car. The **2** here is the rate of acceleration. For smaller numbers, the car accelerates slower, but for larger numbers it accelerates faster.

Example:

How can you adjust the code above so that the car speeds up until it gets to the center of the window and then slows down (i.e., decelerates) so that it stops at the right of the window ?

Algorithm: DecelerateCar

```
1.      speed ← 0
2.      for x locations from 0 to windowWidth by speed {
3.          draw the car at position x
4.          if x < (windowWidth /2) then
5.              speed ← speed + 2
6.          otherwise
7.              speed ← speed – 2
        }
```

2.5 Conditional Iteration

In the above examples, we used variables and loops to show how we can count within an algorithm. In each case there was a fixed number of items to iterate through. However, the situation often arises if real life when the total number of items to iterate through is unknown.

For example, a cashier must be able to repeat the scanning of items from a customer without knowing exactly how many items there will be. In fact, in this case the number of items is not important, only the final cost of the items being purchased is required. In such a situation, the repeated scanning of items will occur until some kind of ***condition*** is satisfied. For example, scanning continues until there are no more items. We will now consider a few more examples that show the need for ***conditional looping*** (i.e., looping that requires some kind of stopping condition).

Example:

Consider the slight variation of the problem in which we want to find the average of a set of grades… but we don't know in advance how many there will be (i.e., **n** is no longer a parameter). How would we write the algorithm ?

```
Algorithm: Average

1.      sum ← 0
2.      repeat until no more grades available {
3.          x ← the next grade
4.          sum ← sum + x
        }
5.      compute the average to be sum / (total number of grades).
6.      print the average.
```

The condition for stopping the loop is when no more grades are available. Depending on where the grades came from, this condition would need to be specified more clearly (e.g., if the user has stopped entering them manually, or if we reached the end of a file, or if some time limit has been reached, etc..).

Both **sum** and **x** are variables here, but there is another "hidden" variable. We need to have the total number of grades in order to compute the average. Where *is* this total ? We need to keep count of the number of grades that have been entered. So, we'll need this variable which starts at 0 and then increases each time we get a new grade.

Here is the more formal version:

```
Algorithm: Average1

1.      sum ← 0
2.      count ← 0
3.      repeat until no more grades are available {
4.          x ← the next grade
5.          sum ← sum + x
6.          count ← count + 1
        }
7.      average ← sum / count
8.      print average
```

This kind of repeat loop is called a ***repeat-until*** loop because it repeats a loop until a certain condition occurs.

Often, it is re-written with the "*until*" part at the bottom as follows:

Algorithm: Average2

```
1.      sum ← 0
2.      count ← 0
3.      repeat {
4.          x ← the next grade
5.          sum ← sum + x
6.          count ← count + 1
7.      } until no more grades are available
8.      average ← sum / count
9.      print average
```

The "until part" is called the ***loop condition***, since it is the part of the algorithm that decides whether or not to continue looping or to stop. Sometimes the word **while** is used to describe the condition for looping as follows:

Algorithm: Average3

```
1.      sum ← 0
2.      count ← 0
3.      while (more grades are available) {
4.          x ← the next grade
5.          sum ← sum + x
6.          count ← count + 1
        }
7.      average ← sum / count
8.      print average
```

This kind of repeat loop is called a ***while*** loop because it repeats the loop as long as (or while) a certain condition still occurs. The *while* loop is more popular in programming than the *repeat-until* loop, although they do the same thing. Notice that the condition is the opposite from the repeat-until. That is, the condition of the "while loop" indicates when to *continue* the loop, whereas the condition of the "repeat-until" loop indicates when to *stop* the loop.

We will make use of the **while** loop construct more often now in our examples, since this is used in JAVA and Processing, whereas the **repeat-until** looping construct is not.

As it turns out, every **repeat** or **for** loop can be expressed in terms of a **while** loop.

For example, recall the **AccelerateCar** algorithm:

Algorithm: AccelerateCar

```
1.      speed ← 0
2.      for x locations from 0 to windowWidth by speed {
3.          draw the car at position x
4.          speed ← speed + 2
        }
```

Here it is re-written using a **while** loop:

Algorithm: AccelerateCarWhileLoop

```
1.      speed ← 0
2.      x ← 0
3.      while (x <= windowWidth)  {
4.          draw the car at position x
5.          speed ← speed + 2
6.          x ← x + speed
        }
```

However, as a “rule of thumb”, a while loop should only be used when there is uncertainty in how many times the loop will occur. That is, you should only use a while loop when the condition to stop the loop is generated from an unexpected event, not when a fixed counter is to be used.

The **while** loop of the **Average3** algorithm, for example, could get its grades one-by-one from the user, who may type-in a special number to stop the process (e.g., -1). That would be an unexpected event to trigger the stopping of the loop. Therefore the while loop would be a good choice. If however, we knew that there were exactly 75 grades **available**, a **for** loop may be used more naturally, asking for grades from 1 to 75 and then stopping automatically.

Here is another example requiring a **while** loop …

Example:

Consider this example that simulates a cashier scanning items at a checkout line in a store. A **while** loop here would fit naturally since there is no way the cashier could know in advance how many items will be scanned ... it could be 1, 10, 50, etc… In real life, the cashier would likely press a particular button (e.g., **total**, or **done**) once all items are scanned.

Try to identify the variables that would be needed by looking for values that will “vary” during the program or values that are needed in multiple parts of the program

Algorithm: CashierSales

```
1.      while (there are more items) {
2.          scan the next item
        }
3.      add the tax to the total
4.      get the payment from the user
5.      compute the change as payment – total - tax
6.      give the change to the customer
```

You should have identified:

- **total cost** - since it changes during the program.
- **tax** - since it is computed in step **3** and used later in step **5**.
- **payment** - since it is received in step **4** and used later in step **5**.
- **change** - since it is computed in step **5**and used later in step **6**.

Likely, however, you may have missed a "hidden" variable. Notice that step **2** is kind of vague. Within that step, we need to add the **price** of the item to the **total cost**. Likely it would make sense to first get the **price** and afterwards add it to the **total** in a second sub step. Here is the more formal code with the variables:

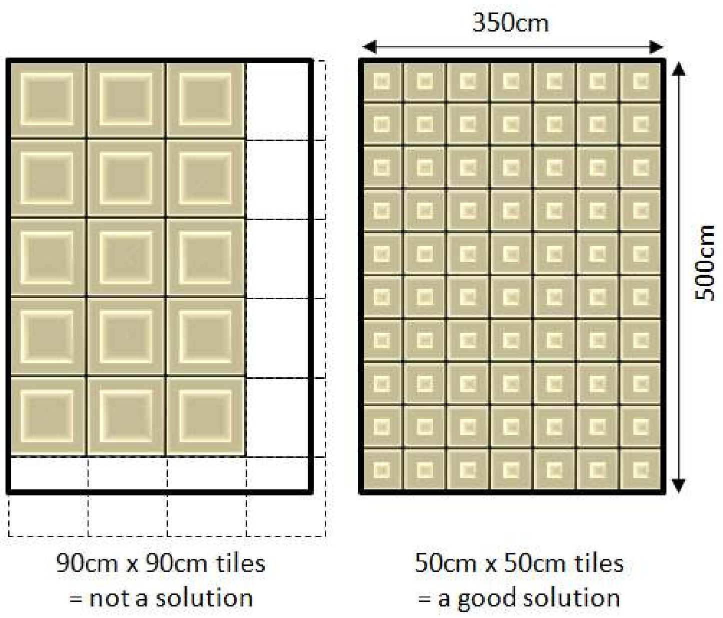
Algorithm: CashierSales

```
1.      total ← 0
2.      while (there are more items) {
3.          price ← result of next scanned item
4.          total ← total + price
        }
5.      tax ← total * 0.13
6.      payment ← payment from customer
7.      change ← payment – total – tax
8.      give back change to customer
```

The **0.13** here is a constant. While it is true that the tax value may change after a few years,(and so it is not a permanent value forever) the value does not change while the program is running and is therefore considered constant.

Example:

Assume that you have a nice rectangular room that measures **W**$_{cm}$ **x H**$_{cm}$. You want to place tiles down on the floor arranged in a grid pattern so that the entire floor is covered. However, you do not want to cut any tiles! Assuming that you can buy pre-cut square tiles of any size, what size of tiles should you buy ?



How do we approach the problem ? Once again, make sure that we understand the problem.

Consider the picture to the right which has a **350**$_{cm}$ **x 500**$_{cm}$ room. In order for the tiles to fit properly, we can only have whole tiles across any row and any column. That means, if we have **R** tiles across a row, then for tiles that are **T**$_{cm}$ **x T**$_{cm}$, then **RxT** must equal exactly **350**. In other words, **350 / T** must be a whole number, not a fraction. So the **T** must divide evenly into **350**. Similarly, **T** must divide evenly into **500** if we are to fit them properly in each column as well.

Certainly we could use 1$_{cm}$ x 1$_{cm}$ tiles in our example above, but that would require **175000** tiles (surely you would not want to lay those down yourself) ! In fact, here are all the possible solutions for our example:

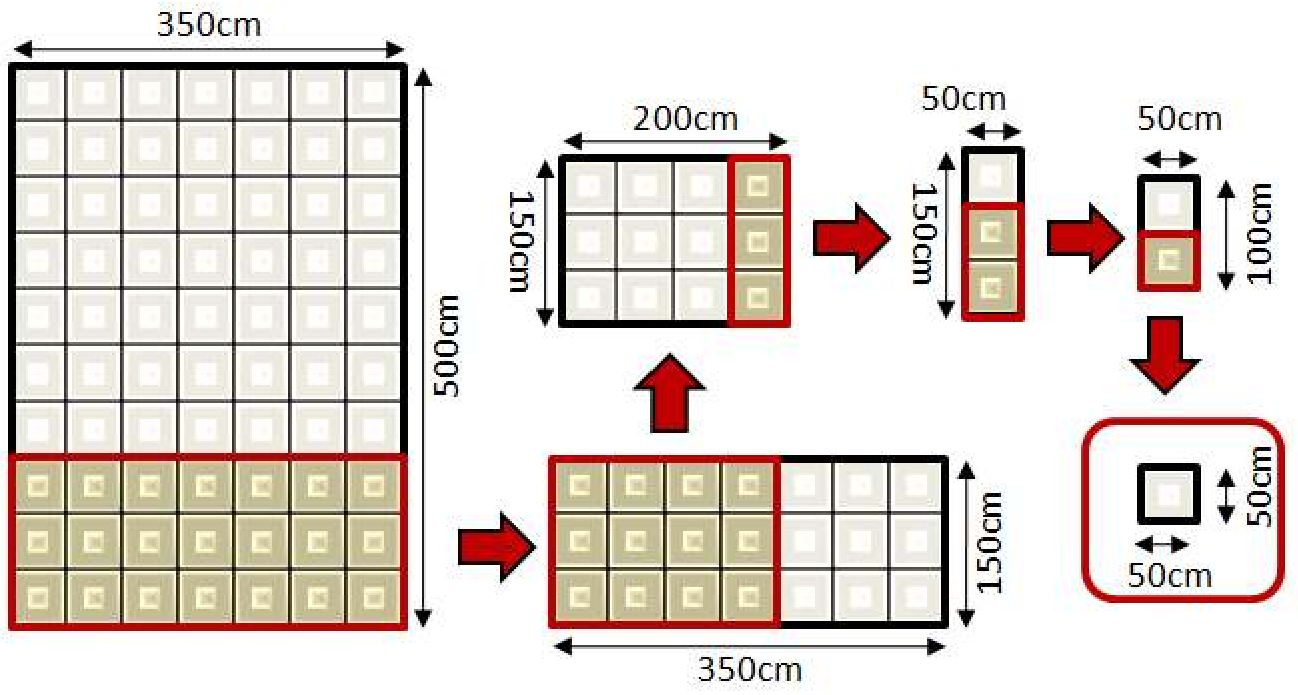
Tile Size	Tiles Required
1$_{cm}$ x **1**$_{cm}$	350 x 500 = **175000**
2$_{cm}$ x **2**$_{cm}$	175 x 250 = **43750**
5$_{cm}$ x **5**$_{cm}$	70 x 100 = **7000**
10$_{cm}$ x **10**$_{cm}$	35 x 50 = **1750**
25$_{cm}$ x **25**$_{cm}$	14x 20 = **280**
50$_{cm}$ x **50**$_{cm}$	7 x 10 = **70**

Likely, the favored solution is the one that requires the least amount of tiles … which is the **50**$_{cm}$ x **50**$_{cm}$ tile solution. The number **50** happens to be the ***greatest common divisor*** (i.e., **GCD**) … or ***greatest common factor*** (i.e., **GCF**) of the numbers **350** and **500**. In fact, the problem that we are trying to solve requires us to find the GCD of our two numbers. Can you think of a simple solution to find that number ?

```
Algorithm: SimpleGCD
          n1, n2:               numbers to which we need to find the GCD

1.        gcd ← minimum of n1 and n2
2.        found ← false
3.        while (not found)  {
4.              if (gcd divides evenly into n1) AND (gcd divides evenly into n2) then
5.                    found ← true
6.              otherwise
7.                    gcd ← gcd - 1
          }
8.        print(gcd)
```

This program will start off with an attempt to see whether or not the smaller number divides evenly into the larger one. If that is true, then we have our answer and the while loop quits. Otherwise, the program keeps subtracting **1** from the potential **gcd** until one is found. Ultimately, this number will keep decreasing to **1**, and that will be a common divisor to any number (although it's the ***least*** common divisor). The program assumes that neither number is zero or negative to begin with.

The above solution will require **300** iterations of the **while** loop (i.e., **gcd** decreases from 350, 349, 348, 347, … down to **50**). There are more efficient solutions. For example, since the **gcd** divides both **350** and **500**, we can see that it is still possible to find the **gcd** by ignoring a large 350_{cm} x 350_{cm} portion of the floor area and concentrating on the remaining area:



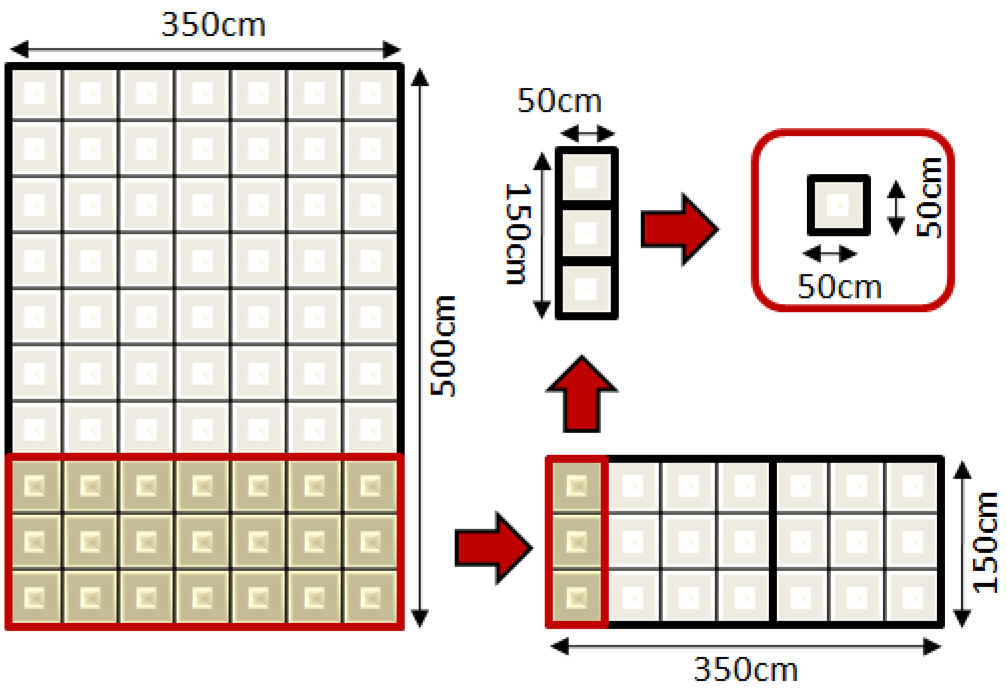
As seen in the diagram, given that we have an **n** x **m** floor area remaining, we can continually extract an **n** x **n** floor area (if **n** < **m**) or an **m** x **m** floor area (if **m** < **n**) until we end up with a remaining floor area in which **n** = **m**. In this case, **n** (or **m**, since they are equal) is the **gcd**.

We can adjust our algorithm do compute the answer in this manner by repeatedly extracting the minimum of the dimensions:

```
Algorithm: BetterGCD
        n1, n2:          numbers to which we need to find the GCD

1.      a ← n1
2.      b ← n2
3.      while (a is not equal to b) {
4.          if a > b then
5.              a ← a - b
6.          otherwise
7.              b ← b - a
        }
8.      print(a)
```

This algorithm produces a better solution … which requires only **5** iterations of the **while** loop!

In fact, it can be improved even further (i.e., only **3** iterations of the **while** loop) by using the modulus operator which takes multiples of the lower dimension away in one step:



We'll leave the details for you to figure out as a practice exercise.

2.6 Control Structures in Processing

As we have seen already through various examples, the need to repeat steps of an algorithm often arises as well as the need to make a decision using **if/otherwise** statements. These are called ***control structures*** in that they specify the flow of control through the program.

The control structures that we have been using in our pseudocode is quite similar to that which is used in Processing (and JAVA). Here, for example is some code that will run in Processing:

```
int grade = 0;
if (grade >= 50) {
    print("Congratulations! ");
    print(grade);
    println(" is a passing grade.");
}
else {
    print(grade);
    println(" is quite low. Oh well, there's always next term.");
}
```

Notice how the **if** statements are not capitalized and that we do not use the word **then**. Also, notice that in place of **otherwise**, we use the word **else**.

The braces **{ }** specify the code that is inside the **if** or **else** part of the conditional statement. If there is only one line within the **if** or **else** body, then the braces are not needed. It is often a good idea to use the braces anyway, even if you have only one line of code because it may prevent you from making some mistakes.

For example, the following code is **not** the same as above:

```
int grade = 0;
if (grade >= 50)
    print("Congratulations! ");
    print(grade);
    println(" is a passing grade.");
else
    print(grade);
    println(" is quite low. Oh well, there's always next term.");
```

The code above will not compile. Since the brackets are missing, the code is assumed to have only one line in the **if** body as follows:

```
int grade = 0;
if (grade >= 50)
    print("Congratulations! ");
print(grade);
println(" is a passing grade.");
else
    print(grade);
println(" is quite low. Oh well, there's always next term.");
```

It then sees the **else** as being out of place. In processing, the **else**, and anything after it will then be ignored completely. So the result would be:

```
0 is a passing grade.
```

In JAVA, the situation is different. You would get a compile error saying: **'else' without 'if'** and the code would not run at all.

An even worse scenario is when Processing/JAVA does not notice the error at all. Consider the following:

```
int grade = 0;
if (grade >= 50)
    print("Congratulations! ");
    print(grade);
    println(" is a passing grade.");
println("All Done.");
```

In the above code, here is the output:

```
0 is a passing grade.
All Done.
```

Clearly this is wrong but the program continues as if nothing bad happened. Also, be careful not to place a semi-colon **;** after the **if** statement brackets:

```
int grade = 0;
if (grade >= 50);
    println("Congratulations! " + grade + " is a passing grade.");
```

In the above code, here is the output:

```
Congratulations! 0 is a passing grade.
```

Why ? Because the semi-colon **;** at the end of the first line tells Processing/JAVA that there is no body for the **if** statement. Thus, the **println(…)** line is outside the **if** statement altogether and is therefore always evaluated.

Recall the GradeToLetter3 algorithm:

```
Algorithm: GradeToLetter3
        grade:      the number grade of a student

1.      if (grade >= 80 and grade <= 100) then
2.            print "A"
3.      otherwise if (grade >= 70) then
4.            print "B"
5.      otherwise if (grade >= 60) then
6.            print "C"
7.      otherwise if (grade >= 50) then
8.            print "D"
9.      otherwise
10.           print "F"
```

Here is the Processing/JAVA equivalent:

```
int grade = ___;  // ignore for now where we got this grade from

if ((grade >= 80) && (grade <=100)) print("A");
else if (grade >= 70) print("B");
else if (grade >= 60) print("C");
else if (grade >= 50) print("D");
else print("F");
```

Of course, we could space this out on a few more lines, but the result would be the same.

Notice the use of **&&**. This is called a ***Boolean operator*** in Processing/JAVA. Here are three useful **boolean** operators:

- **&&** … the same as saying **and**
- **||** … the same as saying **or**
- **!** … the same as saying **not**

You will notice as well that the code used the **>=** operator. This is called a ***logical operator*** that determines whether one number is greater than or equal to another. It takes the two values, compares them, and then determines a **boolean** result of **true** or **false**. Here is the list of all the available logical operators that we can use:

- **<** less than
- **<=** less than or equal to
- **==** equal to
- **!=** not equal to
- **>=** greater than or equal to
- **>** greater than

Notice that when we want to ask if something is equal to another thing we use two equal signs **==**, not one. The single equal sign **=** is only used to assign (i.e., give) a value to a variable.

Recall as well the **DaysInMonth2** algorithm:

```
Algorithm: DaysInMonth2
        month:    a month from 1 to 12

1.      if (month is 2) then
2.              print 28
3.      otherwise if (month is 4) or (month is 6) or (month is 9) or (month is 11) then
4.              print 30
5.      otherwise print 31
```

This could be implemented in Processing/JAVA as follows:

```
int month = ___;  // ignore for now where we got this month from

if (month == 2)
     print(28);
else  if ((month == 4) || (month == 6) || (month == 9) || (month == 11))
     print(30);
else
     print(31);
```

In special cases where there is a list of fixed values that we want to make a decision on (e.g., month is a number from 1 to 12), we can use what is known as a ***switch statement***.

The switch statement has the following format:

```
switch (aPrimitiveExpression) {
    case val1:
        /*one or more lines of JAVA code*/;
        break;
    case val2:
        /*one or more lines of JAVA code*/;
        break;
    ...
    case valN:
        /*one or more lines of JAVA code*/;
        break;
    default:
        /*one or more lines of JAVA code*/;
        break;
}
```

In the above code, **aPrimitiveExpression** is either a primitive variable (e.g., a variable of type **int**, **char**, **float**, etc…) or any code that results in a primitive value. The values of **val**$_1$, **val**$_2$, …, **val**$_N$ must all be primitive constant values of the same type as **aPrimitiveExpression**.

The **switch** statement works as follows:

1. It evaluates **aPrimitiveExpression** to obtain a value (the expression MUST result in a primitive data type, it **cannot** be an object (more on this later)).

2. It then checks the values **val1**, **val2**, …, **valN** in order from top to bottom until a value is found equal to the value of **aPrimitiveExpression**. If none match, then the **default** case is evaluated.

3. It then evaluates the statements corresponding to the **case** whose value matched.

4. If there is a **break** at the end of the lines of code for that **case**, then the **switch** statement quits. Otherwise it continues to evaluate all the successive **case** statements that follow ... until a **break** is found or until no more cases remain.

Here is how we can use a **switch** statement for our **DaysInMonth** code …

```
int month = ___;  // ignore for now where we got this month from

switch(month) {
    case 2: print(28); break;
    case 4:
    case 6:
    case 9:
    case 11: print(30); break;
    default: print(31);
}
```

Note that when the month is **4**, **6**, **9**, or **11**, then the **print(30);** is evaluated. The code is not necessarily much shorter, but it is simpler to read. This is the main advantage of a **switch** statement.

One thing that needs mentioning is that the value of the cases must be **primitive literals**. That is, they cannot be expressions, ranges (nor Strings). Nor can we make use of the logical operators such as **and** and **or**. So these three examples will not work:

```
switch (age) {
    case 1 to 12:  price = 5.00; break;  // Won’t compile
    case 13 to 17: price = 8.00; break;  // Won’t compile
    case 18 to 54: price = 10.00; break; // Won’t compile
    default:    price = 6.00;
}
```

```
switch (name) {
    case "Mark":   bonus = 3; break; // Won’t compile
    case "Betty":  bonus = 2; break; // Won’t compile
    case "Jane":   bonus = 1; break; // Won’t compile
    default:       bonus = 0;
}
```

```
switch (month) {
    case 2:                 print(28); break;
    case 4 || 6 || 9 || 11: print(30); break;  // Won’t compile
    default:                print(31);
}
```

Getting back to the **IF** statement, although they are quite easy to use, it is often the case that students do not **fully** understand how to use **boolean** logic. As a result, sometimes students end up writing overly complex and inefficient code ... sometimes even using an **IF** statement when it is not even required!

To illustrate this, consider the following examples of "BAD" coding style. Try to determine why the code is inefficient and how to improve it. If it is your desire to be a good programmer, pay careful attention to these examples.

Example 1:

```
boolean   male = ...;

if (male == true) {
   println("male");
else
   println("female");
```

Here, the **boolean** value of **male** is *already* **true** or **false**, we can make use of this fact:

```
boolean   male = ...;

if (male) {
   println("male");
else
   println("female");
```

Example 2:

```
boolean   adult = ...;

if (adult == false)
     discount = 3.00;
```

Here is a similar situation as above, but with a negated **boolean**. Below is better code.

```
boolean   adult = ...;

if (!adult) {
     discount = 3.00;
```

Example 3:

```
boolean   tired = ...;

if (tired)
     result = true;
else
     result = false;
```

Above, we are actually returning the identical **boolean** as **tired**. No **if** statement is needed:

```
boolean   tired = ...;

result = tired;
```

Example 4:

```
boolean   discount;

if ((age < 6) || (age > 65))
    discount = true;
else
    discount = false;
```

The discount is solely determined by the **age**. No **if** statement is needed:

```
boolean discount;

discount = (age < 6) || (age > 65);
```

Example5:

```
boolean   fullPrice;

if ((age < 6) || (age > 65))
    fullPrice = false;
else
    fullPrice = true;
```

Just like above, we do not need the **if** statement:

```
boolean   fullPrice;

fullPrice = !((age < 6) || (age > 65));
or …
fullPrice = (age >= 6) && (age <= 65);
```

Now what about the **repeat**, **for** and **while** loops ? They too are control structures. Recall the stair-counting algorithm:

Algorithm: Stairs

```
1.      total ← 0
2.      currentHeight ← 1
3.      repeat n times {
4.          total ← total + currentHeight
5.          currentHeight ← currentHeight +1
        }
6.      print total
```

What would this look like in Processing ? Here is the solution:

```
int total = 0;
for (int currentHeight = 1; currentHeight <= n; currentHeight++) {
   total = total + currentHeight;
}
println(total);
```

Let us discuss this code a bit. Notice that the **total** starts off at 0 and then the **currentHeight** is added to it, just as in our pseudocode, and finally the **total** is printed.

Notice that the **for loop** has what looks like parameters and code within the parentheses **( )** and that it has braces **{ }** just as a function of procedure does. The code within the braces is called the ***loop body***, and it can be any chunk of code which will be evaluated over and over again depending on the code within the parentheses.

Lets break down the code within the paretheses. You may notice that there are two semi-colons that break things into two portions as follows:

(*initializer* ; *loopTest* ; *countExpression*)

Each of these portions is explained here:

- **initializer** – this is usually used to declare and initialize (i.e., set the starting value for) a variable (called the ***loop variable***) which will be used as a counter within the loop. The loop variable can be used anywhere within the **for** loop but not outside of it. In most situations, this counter starts off at 0 or 1, but there are some times when you will use other values.

- **loopTest** – this is any coding expression that results in a boolean result. It is used to determine whether or not to go back into the loop again for another round. Usually, this loop will check if the loop variable has reached some kind of limit. As long as the boolean expression results in **true**, the loop will repeat again.

- **countExpression** – this is a portion of code that is evaluated AFTER each time the loop has completed one iteration (i.e., one round). It is usually used to increase or decrease the value of the loop variable by some value (such as 1), although this not always the case.

So in our example, the **loopTest** checks to see if the **currentHeight <= n**. If it evaluates to **true**, then it keeps looping, otherwise it no longer repeats the loop code. The **loopTest** is checked before the loop starts. If it is false right away, then the entire **for** loop is ignored and never evaluated. Otherwise, the loop is evaluated at least once. At the end each loop iteration (i.e., after the loop body has been evaluated), the **countExpression** is evaluated and then the **loopTest** is performed again in order to decide whether or not to continue another round at the top of the loop.

The **countExpression** has a **++** at the end of the **currentHeight** variable. This is called the ***increment operator***. It has the same result as doing:

currentHeight = **currentHeight** + 1;

This is evaluated AFTER each iteration of the loop and BEFORE the **loopTest** is performed again.

There is also a ***decrement operator*** **--** which has the same result as subtracting **1** from the variable. It is very useful when you are counting down from a number.

Often, in order to keep code simple to read, a programmer will use the letter **i** to represent the loop variable (**i** being short for "*index*"). Here is how our code will look with **i** instead:

```
for (int i=1; i<=n; i++) {
  total = total + i;
}
```

The code looks much simpler. In addition, if the body of a **for** loop has only one line of code in it, then the brace **{ }** characters are not needed:

```
for (int i=1; i<=n; i++)
  total = total + i;
```

Recall as well that we sometimes want to count by more than 1 each time.

For example, the code to accelerate a car increased the x value by a speed component as follows:

Algorithm: AccelerateCar

```
1.      speed ← 0
2.      for x locations from 0 to windowWidth by speed {
3.          draw the car at position x
4.          speed ← speed + 2
        }
```

Here is what the Processing code would look like:

```
int speed = 0;
for (int x=0; x<=width; x=x+speed) {
     drawCarAt(x);  // details left out
     speed = speed + 2;
}
```

A few things to note. First, **width** is a pre-defined parameter in Processing that is set to the width of the window (in pixels). Similarly, there is a **height** parameter set to the height of the window (in pixels). Notice as well now that the **x** value is increased by **speed** instead of by 1 each time.
What would happen if the stopping-expression of a **for** loop evaluated to **false** right away as follows:

```
for (int x=0; x>=width; x++) {
      ...
}
```

In this case, **x** starts at **0** and the stopping condition determines that it is not greater than or equal to **width**. Thus, the loop body never gets evaluated. That is, the **for** loop does nothing … your program ignores it.

A similar unintentional situation may occur if you accidentally place a semi-colon **;** after the round brackets by mistake:

```
for (int x=0; x>=width; x++); {
      ...
}
```

In this situation, JAVA assumes that the **for** loop ends at that semi-colon **;** and that it has no body to be evaluated. In this case, the body of the loop is considered to be regular code outside of the **for** loop and it is evaluated once.

Hence the above code is understood as:

```
for (int x=0; x>=width; x++) {
}
...
```

Alternatively, in Processing, we could have used **while** loops in our solutions as follows:

```
int total = 0;
int currentHeight = 1;

while (currentHeight <= n) {
    total = total + currentHeight;
    currentHeight++;
}
println(total);
```

```
int speed = 0;
int x=0;
while (x <= width) {
     drawCarAt(x);
     speed = speed + 2;
     x = x + speed;
}
```

You should notice that the **currentHeight** and **x** variables are still needed, but is now defined outside the loop. However, it is often the case that a **while** loop waits for some event, as opposed to some counter reaching a known limit. We will do more examples later.

Just as with **for** loops, you should be careful not to put a semi-colon **;** after the parentheses, otherwise your *loop body* will not be evaluated. Usually your code will loop forever because the *stopping condition* may never change to **false**:

```
while (n < 100); {                          // This code will loop forever
      println(n++);
}
```

As with the **if** statements and **for** loops, the braces **{ }** are not necessary when the *loop body* contains a single coding expression:

```
while (n >= 0)
      println(n--);
```

Some students tend to confuse the **while** loop with **if** statements and try to replace an **if** statement with a **while** loop. Do you understand the difference in the two pieces of code below ?

```
if (age > 18)
      discount = 0;
```

✓

```
while (age > 18)
      discount = 0;
```

✗

Assume that the person's age is **20**. The leftmost code will set the discount to **0** and move on. The rightmost code will loop forever, continually setting the discount to **0**.

2.7 Procedures & Functions in Processing

You may recall, from our discussion of abstraction, that it is often a good idea to simplify our overall algorithm by making it higher-level. That means, we could hide details that are unnecessary. For example, if we wanted to create a home scenery, it may make sense to develop a high-level algorithm like this:

```
Algorithm: DrawScenery

1.      draw the house
2.      draw the laneway
3.      draw the car
4.      draw the lawn
5.      draw the trees
```

This is an easily understood algorithm which hides all the unnecessary details of "how" to draw the various things. Recall our program for drawing a simple house:

```
size(300,300);

// Draw the house
rect(100,200,100,100);
triangle(100,200,150,150,200,200);
rect(135,260,30,40);
point(155,280);
```

How could we abstract out and make this a higher-level algorithm ? We could create a **drawHouse()** procedure that will draw the house. Then our program would like simpler like this:

```
size(300,300);
drawHouse();
```

What would the **drawHouse()** procedure look like ? Well in Processing and JAVA, this is the format for declaring a simple procedure:

```
void procedureName () {
    // Write your procedure's code here
}
```

Notice that procedure has **void** at the front. This indicates that there is no value to be returned from the procedure.

The brace characters (i.e., **{ }**) indicate the code's body:

*The **body** of a function or procedure is the code that is evaluated each time that the function or procedure is called.*

So then, our **drawHouse()** procedure would look as follows:

```
size(300,300);
drawHouse();

void drawHouse() {
    rect(100,200,100,100);
    triangle(100,200,150,150,200,200);
    rect(135,260,30,40);
    point(155,280);
}
```

The code itself looks more complicated because nothing looks hidden at all! Actually, the above code, by itself, will not compile in Processing. In general, when writing a program in Processing, all of our program code must lie within a function or procedure of some kind. There are exceptions to this:

- we may declare "global variables" outside a function at the top of our program
- when we are not creating any functions or procedures of our own, Processing will allow us to write a simple sequence of code that does not need to be inside a function or procedure.

So, Processing has provided a useful procedure called **setup()** into which we can place our code. The **setup()** procedure is called one time automatically by Processing whenever we start or restart the program. Here is code that will now compile and run in Processing:

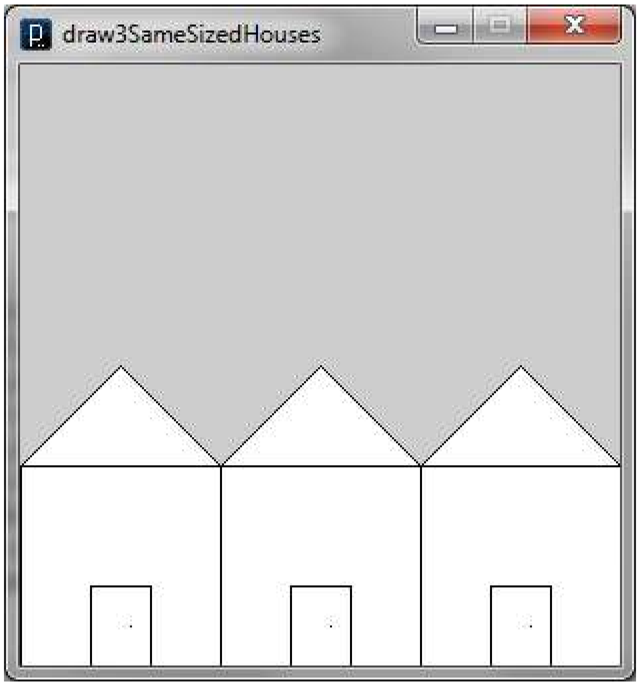
```
void setup() {
    size(300,300);
    drawHouse();
}
```

```
void drawHouse() {
    rect(100,200,100,100);
    triangle(100,200,150,150,200,200);
    rect(135,260,30,40);
    point(155,280);
}
```

Notice that the code is simply made up of two procedures. Again, there seems to be more code, but notice how the main algorithm (i.e., the code in the **setup()** procedure) is much simpler now.

As a side point, in JAVA, C, C#, C++ and similar languages, there is a procedure similar to **setup()** called **main()** … which is called automatically upon program startup. You will see more of this in the follow-up course.

Abstraction, is just one reason for creating a procedure or function. However, efficiency (in the amount of code that is to be written) is another. In order to more fully understand the benefits of creating functions and procedures consider how we can ***efficiently*** draw 3 houses side-by-side. Here is an *inefficient* way to do it:



```
void setup() {
    size(300,300);

    // 1st house
    rect(0,200,100,100);
    triangle(0,200,50,150,100,200);
    rect(35,260,30,40);
    point(55,280);

    // 2nd house
    rect(100,200,100,100);
    triangle(100,200,150,150,200,200);
    rect(135,260,30,40);
    point(155,280);

    // 3rd house
    rect(200,200,100,100);
    triangle(200,200,250,150,300,200);
    rect(235,260,30,40);
    point(255,280);
}
```

The program is inefficient because we are duplicating portions of code. Notice that the only differences in the code is with respect to the values shown underlined in red. Do you notice how these values differ from the 1st house to the 2nd house and from the 2nd house to the 3rd ?

We are actually adding **100** to these values each time that we draw a house. Notice as well that the value is always the **x** coordinate of the shape being drawn … the width and height values do not change. In other words, we are *offsetting* the **x** value of our house by a fixed amount (i.e., **100**) each time we re-draw it.

*An x-offset is the difference by which one graphical object is out of **horizontal** alignment from some fixed horizontal reference (e.g., origin or another object's position).*

*A y-offset is the difference by which one graphical object is out of **vertical** alignment from some fixed vertical reference (e.g., origin or another object's position).*

Notice how we can re-write the code with an x-offset:

```
void setup() {
    int xOffset; // Make a variable to store the offset

    size(300,300);

    // 1st house
    xOffset = 0;
    rect((0+xOffset),200,100,100);
    triangle((0+xOffset),200,(50+xOffset),150,(100+xOffset),200);
    rect((35+xOffset),260,30,40);
    point((55+xOffset),280);

    // 2nd house
    xOffset = 100;
    rect((0+xOffset),200,100,100);
    triangle((0+xOffset),200,(50+xOffset),150,(100+xOffset),200);
    rect((35+xOffset),260,30,40);
    point((55+xOffset),280);

    // 3rd house
    xOffset = 200;
    rect((0+xOffset),200,100,100);
    triangle((0+xOffset),200,(50+xOffset),150,(100+xOffset),200);
    rect((35+xOffset),260,30,40);
    point((55+xOffset),280);
}
```

It seems that we have written **more** code now, so how could this be more efficient ? As written, it is NOT more efficient. However, you should notice that the portion of the code that actually draws each house is exactly the same!

That means, we could create a procedure for drawing the house and simply call it three times as follows:

```
void setup() {
    int xOffset;// Make a variable to store the offset

    size(300,300);

    // draw 3 houses with different x offsets
    xOffset = 0;
    drawHouse(xOffset);

    xOffset = 100;
    drawHouse(xOffset);

    xOffset = 200;
    drawHouse(xOffset);
}
```

What would the **drawHouse()** procedure look like now ? Well, you may notice that we need to pass in a parameter to the procedure in order to indicate the **x** offset. Here is the format for passing in parameters to a function:

> **void** ***procedureName*** (t_1 n_1, t_2 n_2, …, t_k n_k) {
>   // Write your procedure's code here
> }

Each parameter must be declared like a variable (i.e., with a type followed by a name), with commas in between. So, t_1, t_2, …, t_k are the *types* of the parameters to the function while n_1, n_2, …, n_k are the *names* of the parameters. While inside the function, each parameter is available for us to use as … just as we would use any other variable.

So then, our **drawHouse()** procedure would now look like this:

```
void drawHouse(int xOffset) {
   rect((0+xOffset),200,100,100);
   triangle((0+xOffset),200,(50+xOffset),150,(100+xOffset),200);
   rect((35+xOffset),260,30,40);
   point((55+xOffset),280);
}
```

As you can see, the **xOffset** parameter looks just like a variable declaration, but inside the brackets now. It tells the Processing/JAVA compiler to reserve space for this incoming integer value which can be used within the procedure.

Where does **xOffset** get its value ? When we call the procedure. Each time we call it with a different offset.

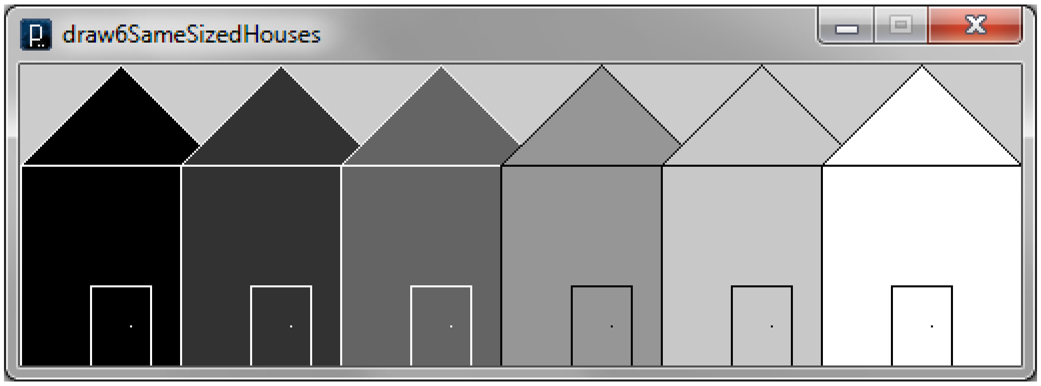
Here is the simplified code:

```
void setup() {
      size(300,300);

      // draw 3 houses with different x offsets
      drawHouse(0);
      drawHouse(100);
      drawHouse(200);
}

void drawHouse(int xOffset) {
      rect((0+xOffset),200,100,100);
      triangle((0+xOffset),200,(50+xOffset),150,(100+xOffset),200);
      rect((35+xOffset),260,30,40);
      point((55+xOffset),280);
}
```

Notice that we no longer need the **xOffset** variable that was declared outside the procedure because we can simply pass in the value for the offset each time we call the procedure.

Example:

Can you write a program that would produce this picture:



What is different from the last program ?

- 6 houses instead of 3
- The size of the window is different … now **500 x 150**
- offset is not 100 anymore, but less (since houses overlap)…**80** ?
- the color of gray changes as the houses are drawn
- the first 3 houses have black border while the last three have white

So, now that we understand the differences, how do we write the code that uses the same **drawHouse()** procedure that we wrote earlier ?

We need to vary the **stroke** color and the **fill** color for each house as follows:

```
void setup() {
    size(500,150);

    stroke(255); // use a white border on everything
    fill(0);drawHouse(0);
    fill(50);drawHouse(80);
    fill(100);drawHouse(160);

    stroke(0); // use a white border on everything
    fill(150);drawHouse(240);
    fill(200);drawHouse(320);
    fill(255);drawHouse(400);
}

void drawHouse(int x) {
    rect(x,50,100,100);
    triangle(x,50,x+50,0,x+100,50);
    rect(x+35,110,30,40);
    point(x+55,130);
}
```

Notice how the **drawHouse()** procedure remains the same, but that the main algorithm differed. How would we change the **drawHouse()** function so that it takes two more parameters that specify the **stroke** and **fill** colors ? Here it is:

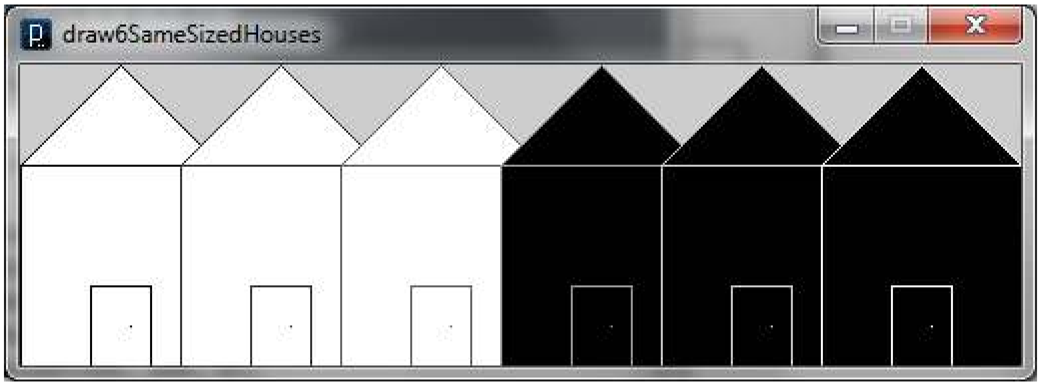
```
void drawHouse(int x, int s, int f) {
    stroke(s);
    fill(f);
    rect(x+0,50,100,100);
    triangle(x+0,50,x+50,0,x+100,50);
    rect(x+35,110,30,40);
    point(x+55,130);
}
```

Notice how we simply indicate two additional parameter types and names in the parameter list to the function and that we make use of these values on the first two lines of the function. How does the simplified setup code now look ?

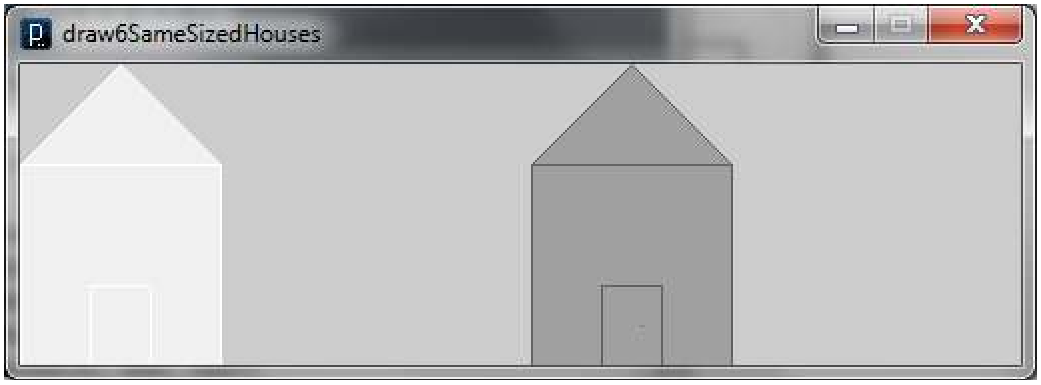
```
void setup() {
    size(500,150);

    drawHouse(0, 255, 0);
    drawHouse(80, 255, 50);
    drawHouse(160, 255, 100);
    drawHouse(240, 0, 150);
    drawHouse(320, 0, 200);
    drawHouse(400, 0, 255);
}
```

Wow. That looks like nice and clean code.

What would happen if we forgot the order of the parameters and mixed the order up between the stroke and the fill. What is we passed the parameters in this order (xOffset, fill, stroke) ?



Or even worse, if we did it in this order (stroke, fill, xOffset) ?



The point is…that lots can go wrong if you mix up the order of your parameters.
But what if we forget to pass in a parameter ? What if you tried **drawHouse(100, 0)** … unintentionally forgetting the fill color ? Well, this would be caught as a compile error indicating:

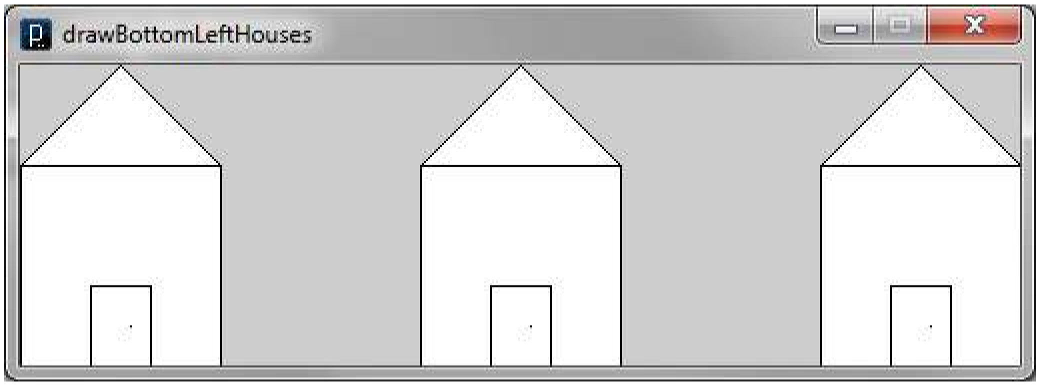
The method drawHouse(int, int, int) … is not applicable for the arguments (int, int)

A similar error would also occur if you passed in too many parameters to the procedure.

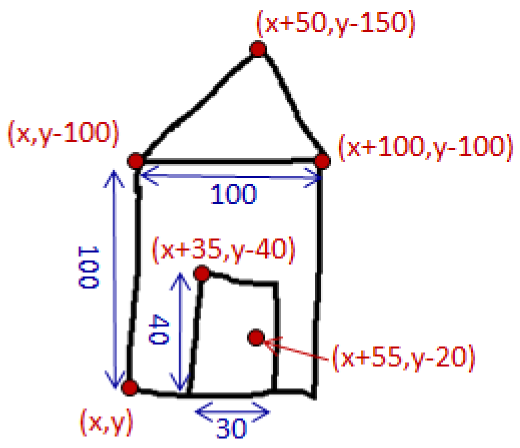
Example:

Adjust the **drawHouse()** procedure to take both an **x** and **y** value representing the bottom-left corner of the house and have the house drawn with respect to that coordinate. For example, if this was the code in the **setup()** method, then the picture shown would result:

```
void setup() {
    size(500,150);
    drawHouse(0, 150);
    drawHouse(200, 150);
    drawHouse(400, 150);
}
```



To do this, we will need to re-compute the coordinate values for our house points with respect to the (x,y) being the bottom left:



Now we can re-write our procedure accordingly to take the extra parameter and adjust the points:

```
void drawHouse(int x, int y) {
     rect(x,(y-100),100,100);
     triangle(x,(y-100),(x+50),(y-150),(x+100),(y-100));
     rect((x+35),(y-40),30,40);
     point((x+55),(y-20));
}
```

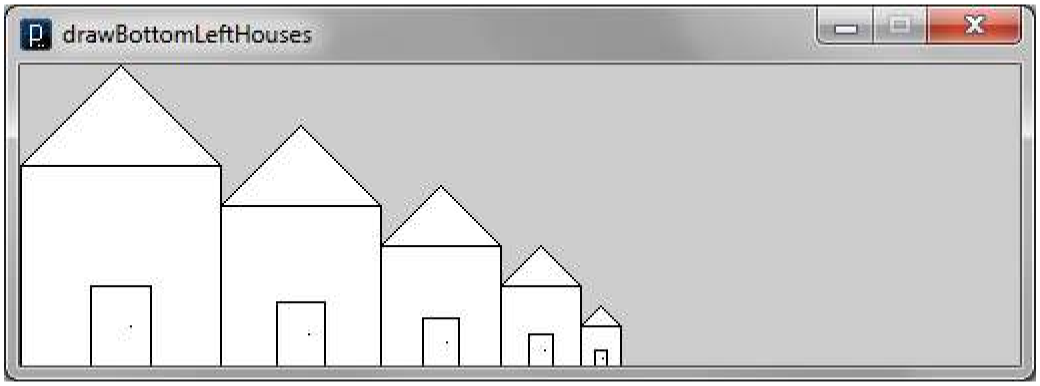
That was not too difficult, but it did require some computations.

Example:

Add a **scale** parameter (i.e., a **float** between 1 and 0) as a third parameter to the **drawHouse()** procedure from the previous example.

Use the scale parameter so that the following code produces the image shown:

```
void setup() {
    size(500,150);

    drawHouse(0, 150, 1);
    drawHouse(100, 150, 0.8);
    drawHouse(180, 150, 0.6);
    drawHouse(240, 150, 0.4);
    drawHouse(280, 150, 0.2);
}
```



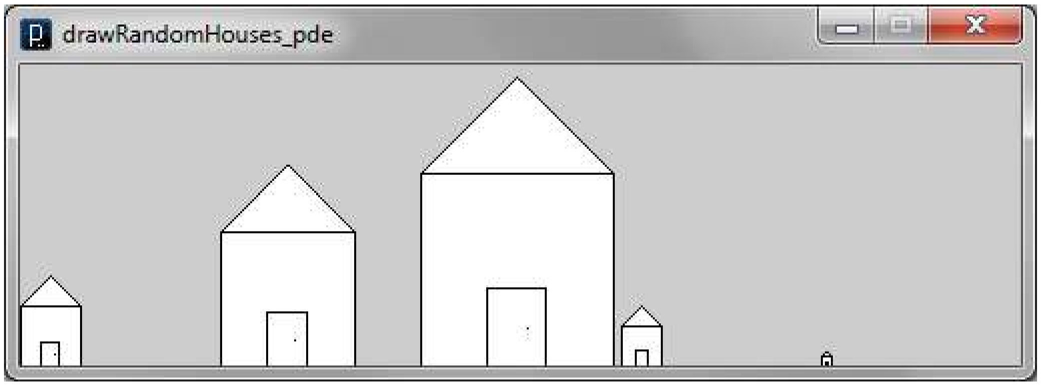
The code is not too difficult, but we must understand how the scale works. The **setup()** method has already adjusted for the position of the bottom-left corner of the houses. All that remains is to ensure that the dimensions are all somehow adjusted by the scale value:

```
void drawHouse(int x, int y, float s) {
    rect(x, y-100*s,100*s,100*s);
    triangle(x,y-100*s,(x+50*s),y-150*s,(x+100*s),y-100*s);
    rect((x+35*s),y-40*s,30*s,40*s);
    point((x+55*s),y-20*s);
}
```

Notice that we simply multiply all dimensions and offsets (i.e., any constant numbers) by the scalar value of **s**.

Example:

What if we wanted to have a random value for the scale so that our houses had different sizes each time we ran the program:



This can be done simply by moving the scale parameter into the procedure and making use of the **random()** function in Processing. The **random(r)** function will return a random float value in the range from **0** to **r**.

Here is how we could adjust the code:

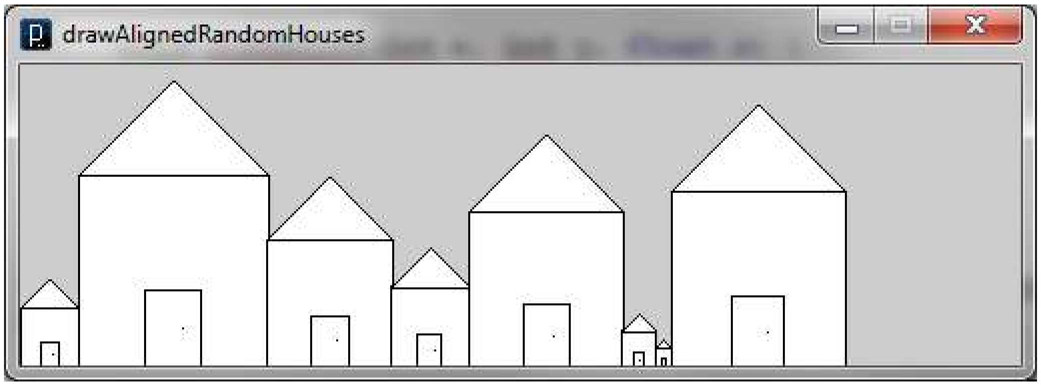
```
void drawHouse(int x, int y) {
    float s;

    s = random(1);
    rect(x, y-100*s,100*s,100*s);
    triangle(x,y-100*s,(x+50*s),y-150*s,(x+100*s),y-100*s);
    rect((x+35*s),y-40*s,30*s,40*s);
    point((x+55*s),y-20*s);
}
```

Notice how the variable **s** is declared within the procedure as a *local variable*. That means that **s** cannot be used outside of the procedure.

Of course, if we don't want our houses to overlap, we should ensure that we adjust our **x** offsets to be at least **100** pixels from one another:

```
void setup() {
    size(500,150);

    for (int x=0; x<=400; x=x+100)
       drawHouse(x, 150);
}
```

Sometimes we need to return a value from our procedure so that we can make use of it in our main program. For example, what would we have to change in order to adjust our previous code so that it packs 8 houses close together according to their scale as follows:



Think of what has changed and develop the algorithm. It is only the x-offset for each house that must vary each time. How much does it vary each time ? It varies according to how much we have scaled the house. For example, in the above image, perhaps the first house had a width of **30**. In that case the 2nd house would have an offset of **30** as its bottom-left corner. Assuming then that the 2nd house had a width of **95**, then the third house would have an offset of **95** from the 2nd house's corner (or **30+95=125** from the left side of the screen). So we can piece this together into an algorithm:

Algorithm: DrawPackedHouses

```
1.      xOffset ← 0
2.      width ← random value
3.      draw the 1st house at xOffset
4.      xOffset ← xOffset + width
5.      width ← random value
6.      draw the 2nd house at xOffset
7.      xOffset ← xOffset + width
8.      width ← random value
9.      etc…
```

You may notice from our previous code that the width of the house is actually **100*s**, where **s** is the randomly chosen scale:

```
void drawHouse(int x, int y) {
    float s;

    s = random(1);
    rect(x, y-100*s,(100*s),100*s);
    ...
}
```

Comparing our algorithm with the code that we already have tells us that step **2** of the algorithm is done as the first couple of lines while we are drawing the house (i.e., within the

drawHouse() procedure). Lines **4** and **7** of the algorithm, however, require the width of the previously drawn house in order to compute the offset for the next house to be drawn.

You should realize that we need to get back the width of the house after we draw it, so that it can be used to compute the offset of the next house. That means, our **drawHouse()** procedure must actually become a ***function*** … in that we now need a value returned from it.

Functions are created the same way that procedures are, but with one exception. Instead of **void**, a function must declare the *type* of the value returned as follows:

t_r ***functionName*** (t_1 n_1, t_2 n_2, …, t_k n_k) {
    // Write your function's code here
}

t_r here is the *return type* of the function:

*A **return type** is the type of the value that is returned from a function.*

So a function is exactly the same as a procedure, except that it must return a value of the type specified as its return type.

In our example, the width of the house (i.e., **100*s**) is the value that must be returned. This is an **int** type. So, here is the function that we need:

```
int drawHouse(int x, int y) {
    float s;

    s = random(1);
    rect(x, y-100*s,100*s,100*s);
    triangle(x,y-100*s,(x+50*s),y-150*s,(x+100*s),y-100*s);
    rect((x+35*s),y-40*s,30*s,40*s);
    point((x+55*s),y-20*s);

    return 100*s;
}
```

Notice the return type (shown underlined in red) and that we use what is called a ***return statement*** at the bottom to indicate what value will be returned as a result of the function call.

The above code, however will not compile. It will return an error saying:

cannot convert from float to int .

It gives this error because the function requires an integer to be returned (i.e., return type is **int**) but we are trying to return **100*s** which is a **float**. We need to convert the return value to an integer.

Processing offers some pre-defined conversion functions for converting between the various data types:

- `int(x)` // converts x into an **int**
- `float(x)` // converts x into a **float**
- `byte(x)` // converts x into a **byte**
- `char(x)` // converts x into a **char**

So in our function, we need to use **return int(100*s)** in order to get the integer that we need as a result. As a side note, however, JAVA does not have such conversion functions. Instead, it uses something called type-casting with different syntax as follows:

- `(int)x` // converts x into an **int**
- `(float)x` // converts x into a **float**
- `(byte)x` // converts x into a **byte**
- `(char)x` // converts x into a **char**

So then, how do we make use of this new function ? Well, we need to use the **width** from the previous **drawHouse()** function call as the **xOffset** for the next house:

Algorithm: DrawPackedHouses2

```
1.      xOffset ← 0
2.      xOffset ← xOffset + drawHouse(xOffset)
3.      xOffset ← xOffset + drawHouse(xOffset)
4.      xOffset ← xOffset + drawHouse(xOffset)
5.      etc…
```

Notice that since the **drawHouse()** call returns the width of the drawn house, we simply keep adding these widths to the **xOffset** to draw each successive house. Here is the Processing code:

```
void setup() {
    size(500,150);

    int xOffset = 0;
    for (int i=0; i<8; i++)
        xOffset = xOffset + drawHouse(xOffset, 150);
}
int drawHouse(int x, int y) {
    float s;

    s = random(1);
    rect(x, y-100*s,100*s,100*s);
    triangle(x,y-100*s,(x+50*s),y-150*s,(x+100*s),y-100*s);
    rect((x+35*s),y-40*s,30*s,40*s);
    point((x+55*s),y-20*s);

    return int(100*s);
}
```

You will be creating many functions and procedures throughout the course. There isn't much more to say about them at this point.

2.8 Math & Trigonometry

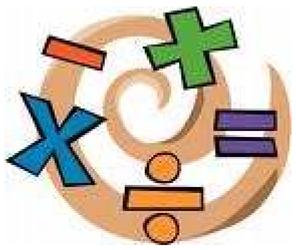
Obviously, a computer can compute solutions to mathematical expressions. We can actually perform simple math expressions such as:

30 + 5 * 2 - 18 / 2 – 2

In such a math expression, we need to understand the order that these calculations are done in. You may recall from high school the **BEDMAS** memory aid which tells you to perform **B**rackets first, then **E**xponents, then **D**ivision & **M**ultiplication, followed by **A**ddition and **S**ubtraction.

So, for example, in the above Processing/JAVA expression, the multiplication ***** operator has preference over the addition **+** operator. In fact, the ***** and **/** operators are evaluated first from left to right and then the **+** and **-**. Thus, the step-by-step evaluation of the expression is:

30 + **5 * 2** - 18 / 2 - 2
30 + 10 - **18 / 2** - 2
30 + 10 - 9 - 2
40 - 9 - 2
31 - 2
29



We can always add round brackets (called ***parentheses***) to the expression to force a different order of evaluation. Expressions in round brackets are evaluated first (left to right):

(30 + 5) * (2 - (18 / 2 - 2))
35 * (2 - (**18 / 2** - 2))
35 * (2 - **(9 - 2)**)
35 * **(2 - 7)**
35 * -5
-175

In Processing/JAVA, it is good to add round brackets around code when it helps the person reading the program to understand what calculations/operations are done first.

Another operator that is often useful is the ***modulus*** operator which returns the remainder after dividing by a certain value.

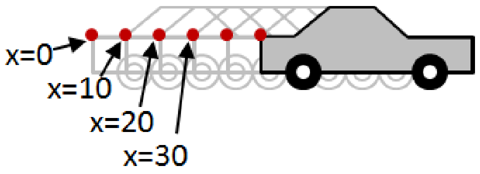
In Processing/JAVA we use the **%** sign as the modulus operator:

```
10 % 2     // results in the remainder after dividing 10 by 2 which is 0
10 % 3     // results in the remainder after dividing 10 by 3 which is 1
10 % 4     // results in the remainder after dividing 10 by 4 which is 2
39 % 20    // results in the remainder after dividing 39 by 20 which is 19
```

Note that using a modulus of 2 will allow you to determine if a number is an odd number or an even number … which may be useful in some applications. Perhaps a more often usage of the modulus operator is to provide a kind of wrap-around effect when increasing or decreasing an integer.

Example:

Recall our algorithm to move a car across the window:



Algorithm: DrawCar

```
1.      for successive x locations from 0 to windowWidth {
2.          draw the car at position x
3.          x ← x + 10
        }
```

What if we wanted the car to drive off the right edge of the window and then re-appear on the left side again ? We could adjust the algorithm as follows:

Algorithm: DrawCarWrapAround1

```
1.      x ← 0
2.      repeat {
3.          draw the car at position x
4.          x ← x + 10
5.          if (x > windowWidth) then
6.              x ← 0
        }
```

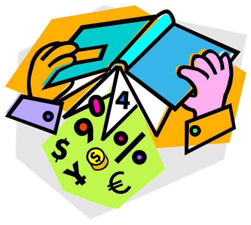
We can eliminate the IF statement and reduce this code simply by using the modulus operator:

Algorithm: DrawCarWrapAround2

```
1.      x ← 0
2.      repeat {
3.          draw the car at position x
4.          x ← (x + 10) % windowWidth
        }
```

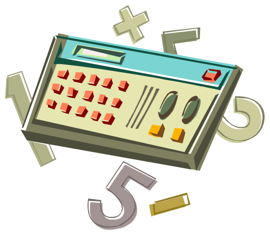
Of course, the above code examples cause the car to re-appear suddenly on the left side of the window. How could we adjust it so that the car "**drives in"** from the left side instead of appearing suddenly ? See if you can figure this out.

Processing actually has many other pre-defined functions that you can use within your programs. Here are just a few of the standard mathematical ones:



- **min(a, b)** – returns the smallest of **a**, **b**, and **c**(optional)
- **max(a, b)** – returns the largest of **a**, **b**, and **c**(optional)
- **round(a)** – rounds **a** up or down to the closest integer
- **pow(a, b)** – returns **a** to the power of **b**
- **sqrt(a)** – returns the square root of **a**
- **abs(a)** – returns the absolute value of **a** (i.e., it discards the negative sign)

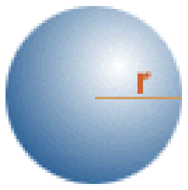
Similar functions are usually available in all programming languages, although their syntax and parameters may vary a little. For example, in JAVA, here is what these functions would be called:



- **Math.min(a, b)** – returns the smallest of **a** and **b**
- **Math.max(a, b)** – returns the largest of **a** and **b**
- **Math.round(a)** – rounds **a** up or down to the closest integer
- **Math.pow(a, b)** – returns **a** to the power of **b**
- **Math.sqrt(a)** – returns the square root of **a**
- **Math.abs(a)** – returns the absolute value of **a** (i.e., it discards the negative sign)

Example:

As an example, consider how to write a program that computes the volume of a ball (e.g., how much space a ball takes up).



How would we write Processing code that computes and displays the volume of such a ball with radius of **25cm** ?

$$\text{Volume} = \frac{4\pi r^3}{3}$$

We need to understand the operations. We need to do a division, some multiplications, raise the radius to the power of 3 and we need to know the value of **π** (i.e., pi).

In Processing, **PI** is defined as a constant with the value 3.14159265358979323846 (in Java, we use **Math.PI**). Here is the simplest, most straight forward solution:

```
int  r = 25;
println(4 * PI * pow(r,3) / 3.0);
```

The following would also have worked, but requires the radius **r** to be duplicated:

```
println(4 * PI * (r*r*r) / 3.0);
```

We could even substitute our own value for **π** :

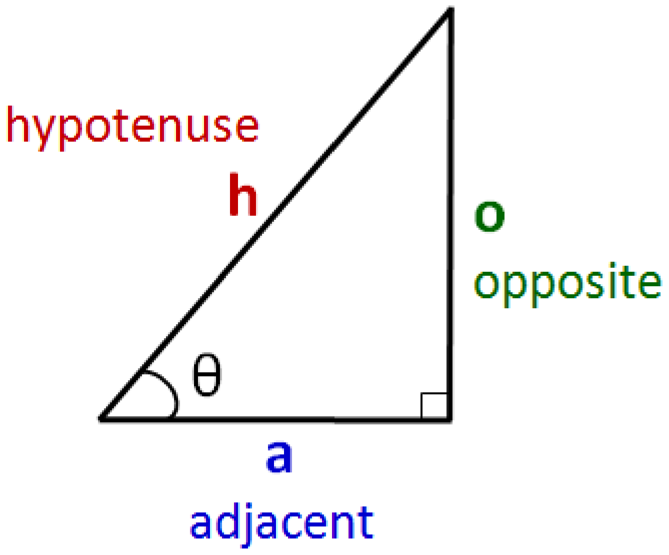
```
    println(4 * 3.14159265358979323846 * (r*r*r) / 3.0);
```

Or we could even pre-compute 4**π**/3 first (which is roughly 4.1887903) :

```
    println(4.1887903 * pow(r,3));
```

The point is that there are often many ways to write out an expression. You will find in this course that there are many solutions to a problem and that everyone in the class will have their own unique solution to a problem (although much of the code will be similar because we will all usually follow the same guidelines when writing our programs).

Besides these basic math functions, there are other VERY useful functions that are often needed in computer science. For example , trigonometric functions are central to computer graphics and for modeling and simulating objects that move around on the screen.



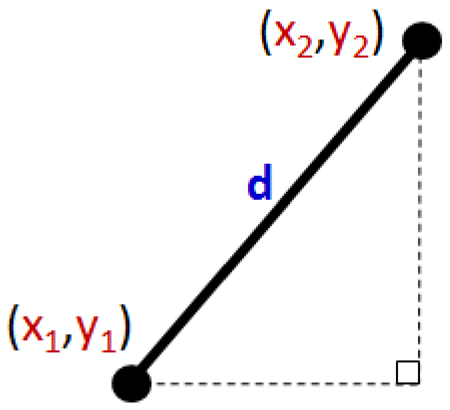
Trigonometry is all based on the angles of a right-angled triangle. Recall that a right-angled triangle has a hypotenuse ... which is the edge opposite to the right angle:

Given one of the other angles, **θ**, of the triangle (either of the ones that is not 90°), we can relate the lengths of the triangle's sides with one other as follows:

sine(θ) = **o**/**h**	→ "soh"
cosine(θ) = **a**/**h**	→ "cah"
tangent(θ) = **o**/**a**	→ "toa"

You may also remember the following formula for calculating the length of **h**:

$$\mathbf{h} = \sqrt{\mathbf{a}^2 + \mathbf{o}^2}$$



What does all of this have to do with computer science ? Well, for one thing, geometry problems are often encountered in computer science and they often require us to determine the distance between two points as follows:

$$\mathbf{d} = \sqrt{(\mathbf{x_2} - \mathbf{x_1})^2 + (\mathbf{y_2} - \mathbf{y_1})^2}$$

In Processing, however, there is a function for doing this:

- **dist(x_1, y_1, x_2, y_2)** – returns the distance between points **(x_1, y_1)** and **(x_2, y_2)**

and in JAVA, here is the function:

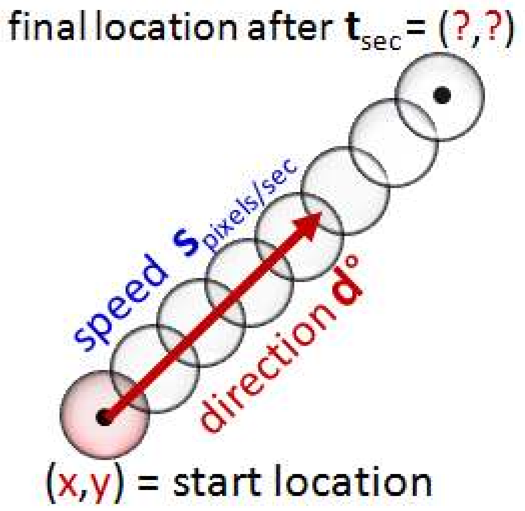
- **Point.distance(x_1, y_1, x_2, y_2)** – returns the distance between points **(x_1, y_1)** and **(x_2, y_2)**

There are countless situations in computational geometry and in computer graphics where knowing the distance between two points is important. We will see examples of this later in the course.

Getting back to the trigonometric functions, they are required in many computational problems as well as simulation and computer graphics problems.

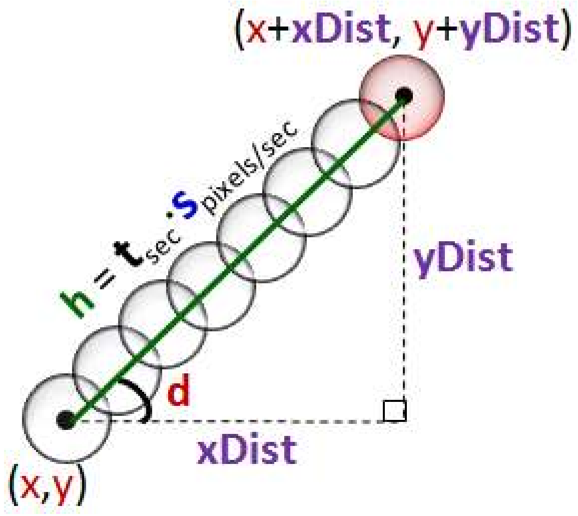
Example:

Consider programming a game in which a ball is moving along at some direction **d** (in degrees with respect to the horizontal axis) and speed **s** (in pixels per second). Given that the ball starts at position (**x**,**y**), where do we redraw the ball after **t** seconds ?



To solve this problem, we simply apply trigonometry. To begin, we need to understand the triangle formed between the start and end location.

The distance between the start and end location cannot be directly computed using the **dist()** function since we do not know the ending location. However, we do know that the distance travelled is (**speed** x **time**). The speed is in pixels per second and the time is in seconds, so the distance travelled, **h**, will have units of pixels.



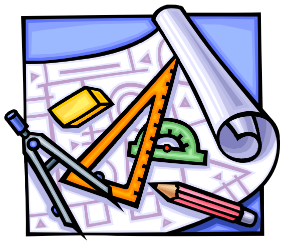
To determine the final location of the ball at time **t**, we just need to determine the amount of movement in the horizontal and vertical directions (i.e., **xDist** and **yDist**). Then we can add those distances to the original (**x**, **y**) location to get the final location.

We can plug in our known trigonometric formulas:

sin(d) = **yDist**/**h** → **yDist** = **h**·**sin**(d)
cos(d) = **xDist**/**h** → **xDist** = **h**·**cos**(d)

And **voila**! We have the final location!!

Whenever doing trigonometry, we must always understand the difference between degrees and radians. Recall that all angles are represented as either **degrees** or **radians**. However, in most programming languages, all trigonometric functions require angles to be specified in radians. Here are the functions in Processing:

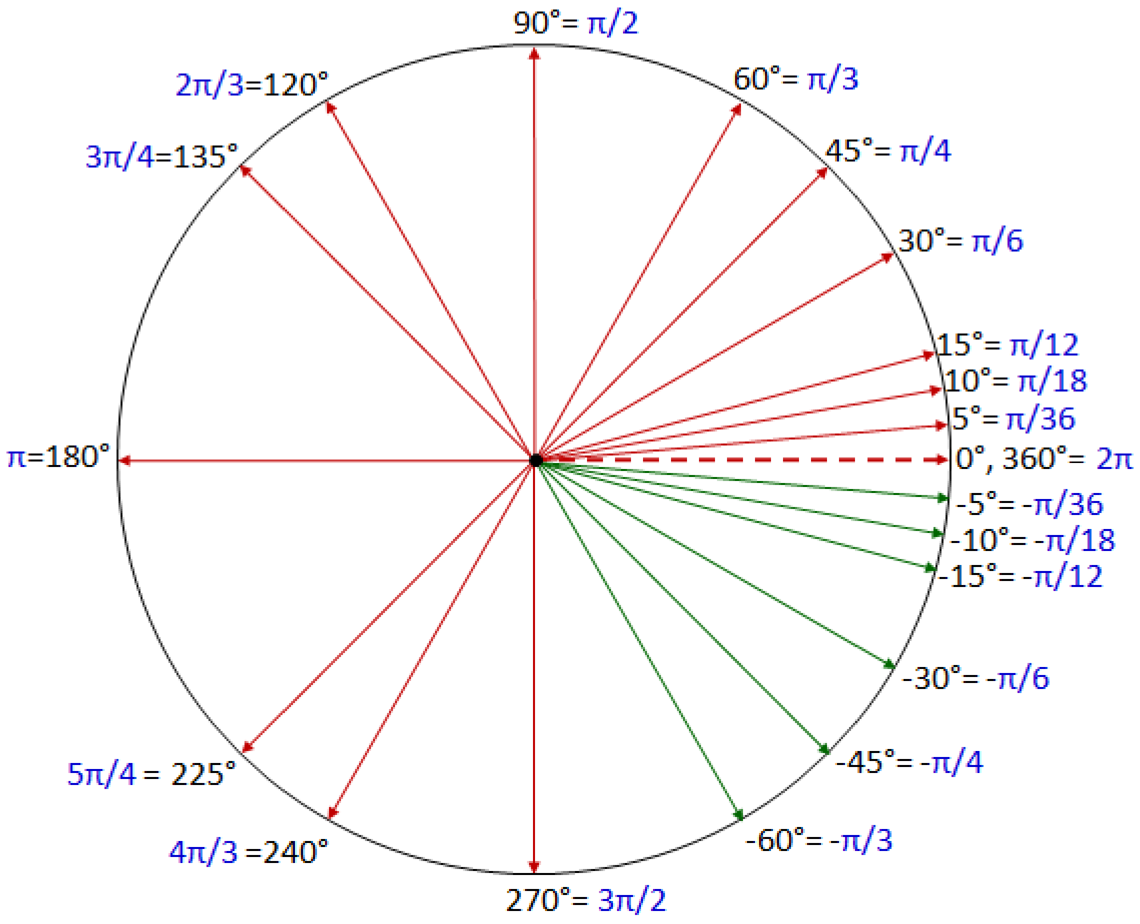


- **sin(a)** – returns the sine of angle **a** (which must be in radians)
- **cos(a)** – returns the cosine of angle **a** (which must be in radians)
- **tan(a)** – returns the tangent of angle **a** (which must be in radians)

Here are the JAVA equivalent functions:

- **Math.sin(a)** – returns the sine of angle **a** (which must be in radians)
- **Math.cos(a)** – returns the cosine of angle **a** (which must be in radians)
- **Math.tan(a)** – returns the tangent of angle **a** (which must be in radians)

Do you remember what radians are ? All degrees and radian values are with respect to a horizontal line that points right … which is the 0° (and 0 radians) angle. A positive increase in angle represents a counter-clockwise spin around a circle, while negative angles represent a clockwise spin. Here is how angles and radians relate to each other:



Just a few common values are shown above. Note also that the negative values all have equivalent positive values as well. So, for example, an angle of 300° (or 5π/3 radians) is the same as the angle -60° (or -π/3 radians).

It is a good idea to understand the above diagram to get used to working with angles. Although the functions all require angles in radians, it is sometimes conceptually easier to store angles as degrees (since it is more intuitive to think in degrees). Because of this, there are conversion functions for converting back and forth between radians and degrees. Here are the conversion functions in Processing:

- **degrees(r)** – returns the degree value for radian angle **r** (computed as 360***r**/2π)
- **radians(d)** – returns the radians value for degree angle **d** (computed as 2π***d**/360)

Here are the JAVA equivalent functions:

- **Math.toDegrees(r)** – returns the degree of angle **r** (which is in radians)
- **Math.toRadians(d)** – returns the radians of angle **d** (which is in degrees)

One more important pre-defined function in most programming languages is the **random** function. In computer science it is often necessary to use random numbers in order to provide variety in our program. For example,

- If we are simulating a colony of ants roaming around on the screen, if we want the ants to seem realistic in their movements, there is a certain degree of unpredictability that must be allowed in the way they move. If, for example, the ants always moved in the same patterns, the simulation would not look realistic.

- If we are testing our program to see if it behaves with unpredictable user input, we may need to generate random data or supply data at random intervals to test the timing of our program. Some algorithms in computer science are based on data that is assumed to be in truly random order.

Obtaining a truly random number is difficult with a computer which is based on 1's and 0's stored in memory. The problem of generating truly random numbers is an open area in computer science that is still being studied. However, many languages supply functions that produce what is called ***pseudo-random*** numbers. That is, the numbers "seem" random, but are actually a fixed sequence of numbers based on some starting point (called the ***seed***) and some function that is applied to that seed successively. Some random number generators simply use the computer's current clock value (i.e., time of day in milliseconds) as a means of obtaining the seed or computing the next random number.

Anyway, there is no need for an in-depth discussion on random number generators. All that is important is that you know that there is a function in Processing that computes a pseudo-random floating point number in the range of 0 to **n**:

- **random(n)** – returns a random floating point number **x** such that 0.0 ≤ **x** < **n**.

So, to get a random number from 0.0 to 99.9999999999, we would call `random(100)`. We can "adjust" the result of this function to obtain a number in any range that we want.
For example, if we wanted an integer in the range from 15 to 67, inclusively, we would do this:

```
int(random(53)) + 15       // where 53 = 67 - 15 + 1
```

In JAVA, the random number generator is different:

- **Math.random()** – returns a random floating point number **x** such that 0.0 ≤ **x** < 1.0.

It always generates a random number from 0.0 to 0.999999999 and if we want to get it within a certain range we need to do additional multiplications:

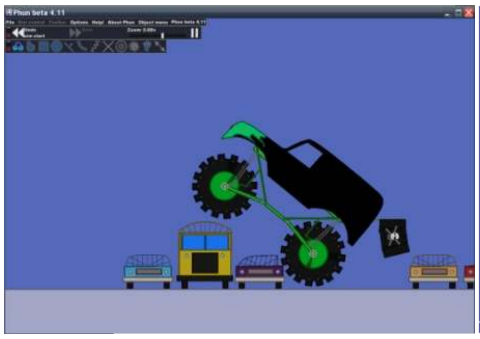
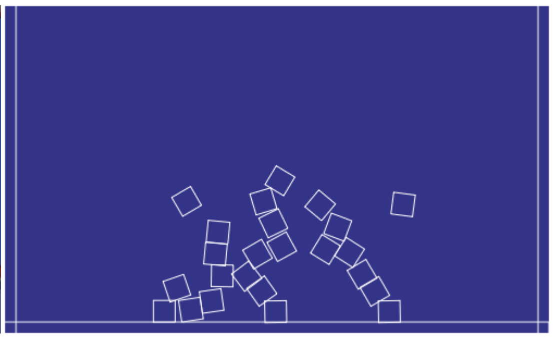
```
100 * Math.random()                  // from 0.0 to 99.99999999
(int)(53 * Math.random()) + 15       // from 15 to 67
```

Chapter 3

Simulation and User Interaction

What is in This Chapter ?

This chapter explains the concepts behind very simple **computer simulations**. All examples are explained within a graphical context. The idea of a processing loop is explained and how various simple 1D and 2D motion-related simulations can be programmed. Next, user interaction is explained within the context of **event handling** within an **event-loop**. Examples are given showing how to handle **mouse-related events**.

3.1 Simulation

Computers are often used to simulate real-world scenarios. Here is the wikipedia definition:

> *A* ***computer simulation*** *is an attempt to model a real-life or hypothetical situation on a computer so that it can be studied to see how the system works. By changing variables, predictions may be made about the behaviour of the system.*

Computer simulation has become a useful part of modeling many natural systems in the areas of physics, chemistry, biology, economics, social science and engineering.

Video games are prime examples of simulation, where some real (or imaginary) life situations are simulated in a virtual world. As time progresses, video games are becoming more life-like as graphics and physical modeling become more precise and realistic.

Other places that require computer simulation are:

- network traffic analysis
- city/urban simulation
- flight/space/vehicle/medical simulators
- disaster preparedness simulations
- film and movie production
- theme park rides
- manufacturing systems

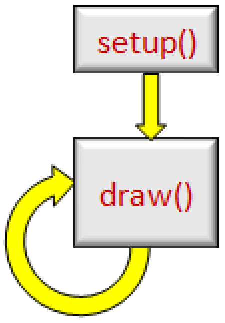
There is much to know about simulation, enough material to fill a course or two. In this course, however, we will just address two basic categories of simulations:

- Running a simulation to find an answer to a problem
- Virtual simulation (i.e., animation) of a real world scenario

When simulating, usually there is some kind of ***initialization*** phase, followed by a ***processing loop*** that continuously processes and (possibly) displays something on the screen.

Think of a simulation as a store. The ***initialization*** phase corresponds to the work involved in getting the store ready to open in the morning (e.g., tidying up, putting out signs, stocking the shelves, preparing the cash register, unlocking the door). The ***processing loop*** phase corresponds to the repeated events that occur during the day (i.e., dealing with customers one-by-one) until the store is to be closed.

In Processing, the **setup()** procedure is called once at the beginning of your program and it represents the initialization phase of your program. It is also used as the main entry point of your program (i.e., your program starts there).

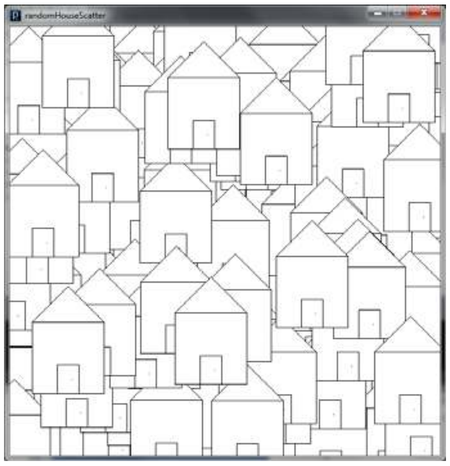


The processing loop phase is accomplished via the **draw()** procedure … which is called repeatedly when your program is running. It is useful for performing repeated computations and displaying information as well as interacting with the user (e.g., via the mouse). By default, the **draw()** procedure is called directly after the **setup()** procedure has been evaluated. Your program does not need to have a **draw()** procedure, but if it is there, your program will call it.

Example:

Recall the **drawHouse()** method that we wrote earlier. Here is how we could repeatedly draw houses at random locations on the screen:

```
void setup() {
    size(600,600);
}

void draw() {
    drawHouse(int(random(width))-50,
              int(random(height))+50);
}

void drawHouse(int x, int y) {
    rect(x, y-100,100,100);
    triangle(x,y-100,(x+50),y-150,(x+100),y-100);
    rect(x+35,y-40,30,40);
    point(x+55,y-20);
}
```

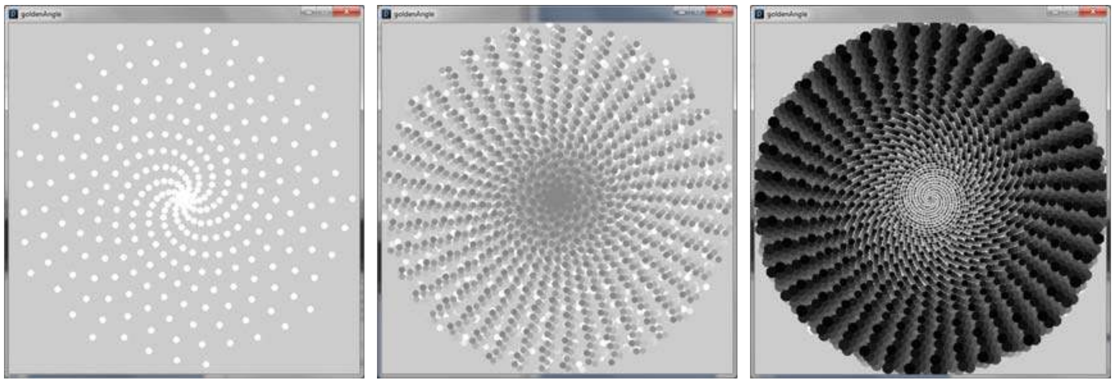


The program represents the simplest kind of simulation … that of simply displaying something in an endless loop. Notice that the **draw()** procedure simply calls **drawHouse()** with a random **x** value from 0 to width=600 and a random **y** value from 0 to height=600 as well. Remember, the **draw()** method is called repeatedly, so the program will continually (and endlessly) draw the house at random positions.

Of course, we could vary the above program to make random sized houses with random colors or we could have it draw the houses along a path (e.g., spiral). Regardless of what we are drawing and how we are drawing it, the programs will all have the same notion of a repeated **draw()** loop.

Example:

Here is a more computational example that displays a spiraling sequence of circles.



You may notice a few differences in the code from the previous example:

```
int     radius;       // distance from center to draw from
float   angle;        // angle to draw at
int     grayLevel;    // color to display each round of spirals

void setup() {
   size(600,600);
   radius = 0;
   angle = 0;
   grayLevel = 0;
}

void draw() {
   stroke(255-grayLevel, 255-grayLevel, 255-grayLevel);
   fill(255-grayLevel, 255-grayLevel, 255-grayLevel);
   ellipse(int(cos(angle)*radius)+width/2,
           int(sin(angle)*radius) + height/2,
           10, 10);

   angle = angle + 137.51;
   radius = (radius+1) % 300;
   if (radius == 0)
       grayLevel = (grayLevel + 10) % 255;
}
```

There are three variables declared at the top of the program. These are declared outside the **setup()** and **draw()** procedures because they are used in both procedures. Hence, they are global variables. The **radius** represents the distance (from the center) that we are drawing the circles at, while the **angle** represents the angle that we are drawing them at. The **grayLevel** indicates the color that the spirals are drawn at, which begins with white and gets darker for each round of spirals.

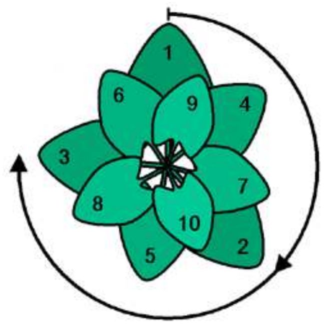
You may notice that the **draw()** procedure first draws the circle (with appropriate gray-level fill and border) and then simply updates the angle and radius for the next time that the **draw()** procedure is called. The **%** operator is the ***modulus operator*** that gives the remained after dividing by a specified number. The modulus operator is great for ensuring that an integer does not exceed a certain value but that it begins again at 0. The following two pieces of code do the same thing:

`radius = (radius + 1) % 255;`	`radius = radius + 1;`
	`if (radius >= 255)`
	`   radius = 0;`

For example if **x** was initially 0 and then we did **x = (x + 1)%5**, here would be the values for **x**:

0,1,2,3,4,0,1,2,3,4,0,1,2,3,4,0,1,2,3,4,0,1,2 … etc..

The angle of **137.51** is called the ***golden angle*** as it is found in nature as the ideal angle for producing spirals as shown in the design of seashells, flower petals, etc.. The angle is ideal as it minimizes overlap during multiple rounds of spiraling.



While this example illustrates the ability to vary the computational parameters during the processing loop of the simulation, it really does not serve any particular purpose other than to produce a nice picture. Now let us look at an example that actually attempts to compute something interesting:

Example:

It is often the case that we need to compute an answer to some problem in which the parameters are complex and/or uncertain. In some situations, it may be unfeasible or impossible to compute an exact result to a problem using a well-defined and predictable algorithm. There is a specific type of simulation method that is well-suited for situations in which you need to make an estimate, forecast or decision where there is significant uncertainty:

> *The **Monte Carlo Method** uses randomly generated or sampled data and computer simulations to obtain approximate solutions to complex mathematical and statistical problems.*

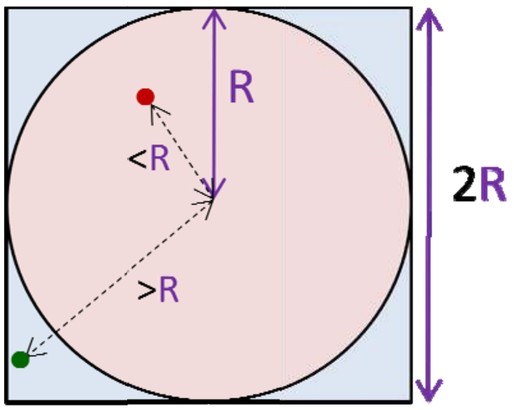
There is always some error involved with this scheme, but the larger the number of random samples taken, the more accurate the result.

The simplest example that is used to describe the Monte Carlo method is that of computing an approximation of **π** (i.e., **pi**). **π** is a mathematical constant whose value is the ratio of any circle's circumference to its diameter. It is approximately equal to **3.141593** in the usual decimal notation. **π** is a very important number in math and computer science as it relates to many trigonometric functions and geometric algorithms and is used in graphics and animation.

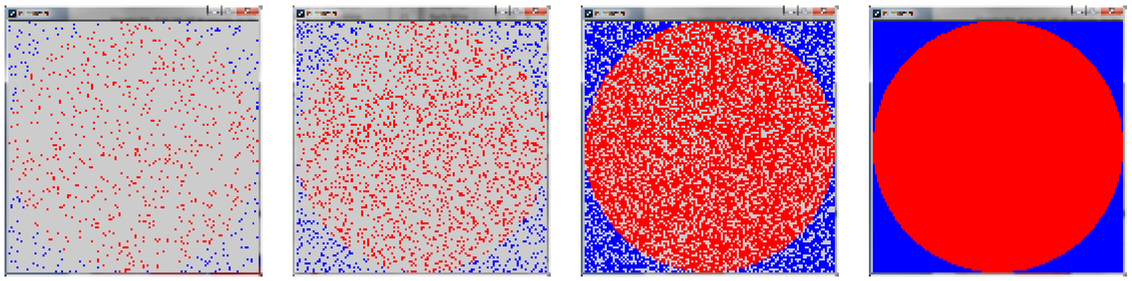
The value of π can be approximated using a Monte Carlo method based on this principle:

> **Given a circle inscribed in a square (i.e., the largest circle that fits in the square), the ratio of the area of the circle to that of the square is π / 4.**

Knowing this, if we can get an estimate for the area of the circle as well as the area of the square, then we can find an approximation for **π / 4**, of course then multiplying by **4** to get an approximation for **π**.



We can estimate the area of the square and circle by uniformly scattering some points throughout the square. Some will lie within the circle, some will lie outside the circle. The more points that we add, the better the approximation of the area, as the whole square and circle will eventually be covered as time goes on.



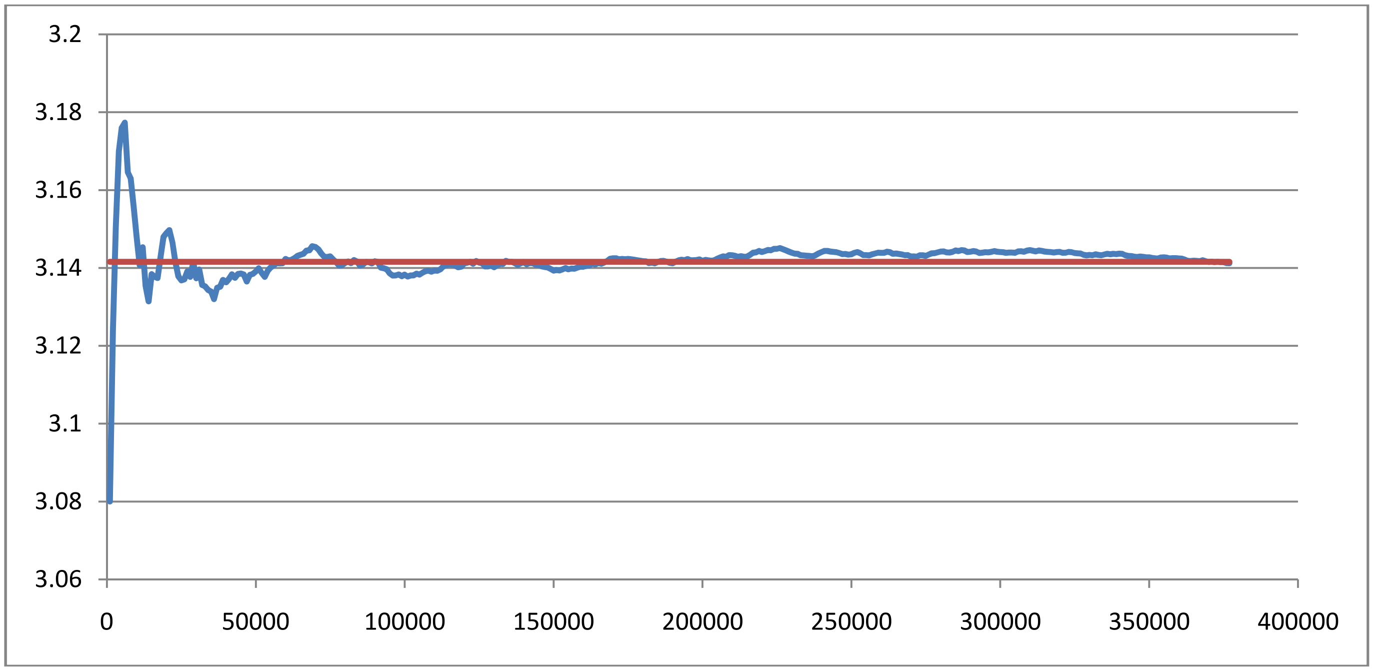
3.12546 3.13725 3.14465 3.14144

Here is the algorithm, assuming that the circle and square are centered at point **(R,R)**:

```
Algorithm: ComputePi
        R:                the radius of a circle and ½ width & ½ height of a square

1.      pointsInCircle ← 0
2.      pointsInSquare ← 0
3.      repeat for a user-chosen amount of iterations {
4.          x ← random value from 0 to 2R-1
5.          y ← random value from 0 to 2R-1
6.          pointsInSquare ← pointsInSquare + 1
7.          if ((the distance from (x,y) to (R,R) < R) then
8.                  pointsInCircle ← pointsInCircle + 1
        }
9.      print (pointsInCircle / pointsInSquare * 4)
```

We can stop the loop at any time. As the loop goes on, however, the algorithm will slowly convergence to a better approximation as more data points are sampled. If the points are purposefully chosen only around the center of the circle, they will not be uniformly distributed, and so the approximation will be poor. An approximation will also be poor if only a few points are randomly chosen throughout the whole square. Thus, the approximation of **π** will become more accurate both with more points and with their uniform distribution.

Here is an example showing how the approximated value (blue) will converge towards the optimal value (solid red line).



In processing, we can write a program that shows us visually what is happening by drawing each point that is randomly chosen. For those inside the circle we can draw them as red and those outside as blue. We can use the **draw()** procedure as our **repeat** loop from our above algorithm since it repeats "forever" (or until the program is manually stopped).
Here is the corresponding Processing code:

```
int     pointsInCircle = 0;
int     pointsInSquare = 0;
int     R = 250;

void setup() {
  size(2*R,2*R);
}

void draw() {
  int x, y;
  x = (int)random(2*R);
  y = (int)random(2*R);
  pointsInSquare = pointsInSquare + 1;
  if (dist(x,y,R,R) < R) {
      pointsInCircle = pointsInCircle + 1;
      stroke(255,0,0);  // red
  }
  else
      stroke(0,0,255);  // blue
  point(x,y);
  println("PI estimated to: " + (double)pointsInCircle/pointsInSquare*4);
}
```

Notice that the code is very similar to the pseudocode. The **random(2*R)** function returns a random number from **0.0** to **2R-1**. This is a floating point number, and so we need to typecast to (**int**) in order to store them in variables **x** and **y**. Alternatively, we could have set the types of **x** and **y** to be **float** instead of **int**. The **dist(x,y,R,R)** function is a pre-defined routine that computes and returns the distance (in pixels) from point **(x,y)** to **point (R,R)**. Notice as well the use of (double) in the code. This ensures that the calculations are done as doubles, and not as integers, otherwise the result would always be **0**.

Each time through the draw loop, the code adds one new point It will take a long time for this code to produce a reasonable approximation for PI. We could add a **for** loop in the **draw()** procedure to add many points each time such as 100 or a 1000 … in order to speed up the approximation process.

Here is the adjusted code.

```
double  pointsInCircle = 0;
double  pointsInSquare = 0;
int     R = 250;

void setup() {
  size(2*R,2*R);
}

void draw() {
  float x, y;

  for (int i=0; i<1000; i++) {
    x = random(width);
    y = random(height);
    pointsInSquare = pointsInSquare + 1;
    if (dist(x,y,R,R) < R) {
      pointsInCircle = pointsInCircle + 1;
      stroke(255,0,0);  // red
    }
    else
      stroke(0,0,255);  // blue
    point(x,y);
  }

  println("PI estimated to: " + pointsInCircle/pointsInSquare*4);
}
```

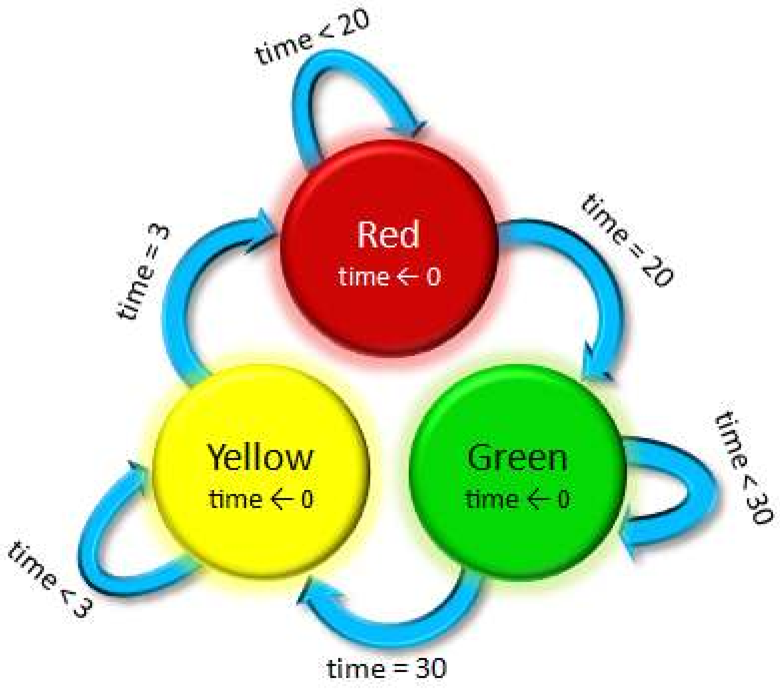
Example:

In the real world we objects have what is known as ***state***. The state of an object is normally considered to be some condition of the object with respect to a previous state. For example, a light bulb is considered to be in a "*working*" state when we buy it, but if we smash it on the ground it would then be in a "*broken*" state.

How can we simulate a traffic light ? It should have 3 states ... RED, GREEN and YELLOW. Assume that the traffic light starts in a RED state and that we want it to cycle continuously between these states. We will assume that the light remains RED for 20 seconds, then GREEN for 30 seconds, then YELLOW for 3 seconds. To simulate the traffic light, it is good to think of it as a state machine.

A **state machine** *is any device that stores the status of something at a given time and can operate on input to change the status and/or cause an action or output to take place for any given change.*

We can then draw a ***state diagram*** to show how the traffic light changes from one state to another as time goes by:



Notice how the state changes from RED to GREEN only when the time has reached 20 seconds. Note as well that inside the state of GREEN, we reset the time counter to 0 so that we can count 30 seconds again in order to decide when to switch to the yellow state.

Assuming that we store the state of the traffic light as a string, how can we write the pseudocode to simulate the light ?

Algorithm: TrafficLight

```
1.  state ← "Red
2.  elapsedTime ← 0
3.  repeat {
4.      if (state is "Red") and (elapsedTime is 20) then {
5.          state ← "Green"
6.          elapsedTime ← 0
        }
7.      otherwise if (state is "Green") and (elapsedTime is 30) then {
8.          state ← "Yellow"
9.          elapsedTime ← 0
        }
10.     otherwise if (state is "Yellow") and (elapsedTime is 3) then {
11.         state ← "Red"
12.         elapsedTime ← 0
        }
13.     elapsedTime ← elapsedTime + 1
    }
```

The code correctly counts properly-proportioned time for each state of the traffic light. However, on a real computer, the **repeat** loop would run much faster than once per second. We would need to slow the whole simulation down so that the elapsed time increases only once per second. In Processing, the **repeat** loop would be represented by the **draw()** procedure which repeats indefinitely. The **frameRate(1)** function sets the re-draw rate of the **draw()** procedure so that it gets called once per second. This will allow the traffic light to operate at the correct speed.

```
String  state;// either red, green or yellow
int     time; // time elapsed since state changed

void setup() {
  state = "Red";
  time = 0;
  frameRate(4);
}

void draw() {
  if ((state == "Red") && (time == 20)) {
    state = "Green";
    time = 0;
    println(state);
  }
  else if ((state == "Green") && (time == 30)) {
    state = "Yellow";
    time = 0;
    println(state);
  }
  else if ((state == "Yellow") && (time == 3)) {
    state = "Red";
    time = 0;
    println(state);
  }
  time++;
}
```

For some fun, try writing code to draw the traffic light as it changes state.

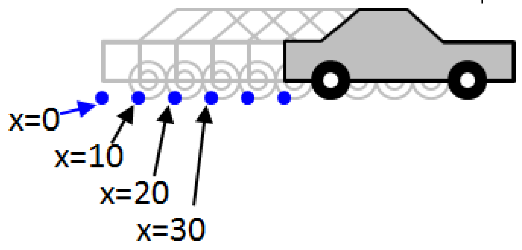
3.2 Simulating Motion

For visually-appealing simulations, it is often necessary to show one or more objects moving on the screen. Such is certainly the case in the area of game programming. We will discuss here some code for doing very simple motion.

Example:

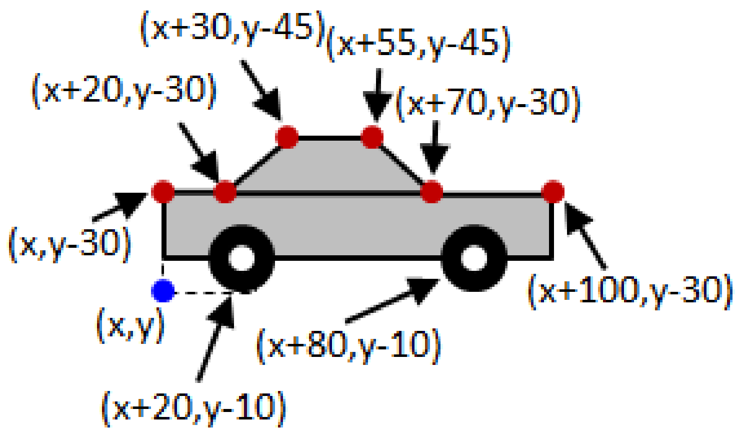
Recall the algorithm that we wrote for moving a car across the screen:

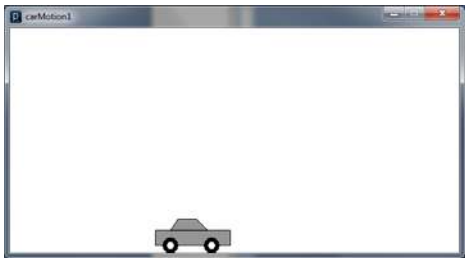
```
Algorithm: DrawCar
        windowWidth:            width of the window

1.        for successive x locations from 0 to windowWidth {
2.            draw the car at position x
3.            x ← x + 10
          }
```

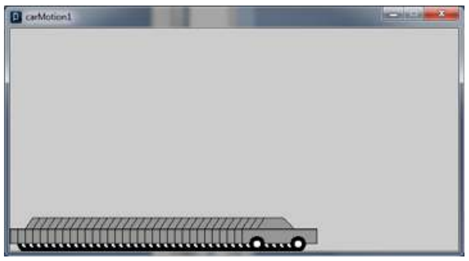


To do this in processing, the code would look like this:

```
int  x, y;

void setup() {
  size(600,300);
  x = 0;
  y = 300;
}

void draw() {
  background(255,255,255);
  drawCar(x);
  x = x + 10;
}

void drawCar(int x) {
  // Draw the body
  fill(150,150,150);  // gray
  rect(x, y-30, 100, 20);
  quad(x+20,y-30,x+30,y-45,x+55,y-45,x+70,y-30);
  // Draw the wheels
  fill(0,0,0);  // black
  ellipse(x+20, y-10, 20, 20);
  ellipse(x+75, y-10, 20, 20);
  fill(255,255,255); // white
  ellipse(x+20, y-10, 10, 10);
  ellipse(x+75, y-10, 10, 10);
}
```







The code above shows the car travelling from the left side of the screen to the right. Notice that the **background()** procedure is called in order to erase the background (makes it white) before drawing the car each time. If we did not do this step, then the previous car positions would be visible.

The code above, however, does not stop the car at the edge of the screen. In order to stop the car, we need to do one of two things:

1) either stop changing the **x** value so that the car is redrawn at the same spot
2) stop the looping

The first solution is easy. We just need to change the last line of the **draw()** procedure to check if we have reached the end and only update when we are not there yet:

```
if (x+100 < width)
    x = x + 10;
```

Notice that we check for the position of the front bumper of the car (i.e.**, x + 100**), not the back bumper (i.e., **x**). Recall as well that **width** is a pre-defined variable that represents the width of the window.

The other option is to stop the loop. There are two ways. First, we can simply call the **exit()** procedure within the draw() procedure in order to quit the program. However, this will cause our program to stop and the window will close. The 2nd way is to call **noLoop()** which temporarily disables the **draw()** procedure until **loop()** is called that will begin calling the **draw()** procedure again:

```
void draw() {
   background(255,255,255);
   drawCar(x);
   x = x + 10;
   if (x+100 > width)
       exit();
}
```

```
void draw() {
   background(255,255,255);
   drawCar(x);
   x = x + 10;
   if (x+100 > width)
       noLoop();
}
```

Example:

The above example showed our car moving rather quickly across the screen. If we adjusted the increment from 10 to a smaller value, the car would move much slower.

Recall our algorithm for accelerating the car until it reaches the middle of the window and then decelerating until it reached the right side of the window again:

```
1.      speed ← 0
2.      for x locations from 0 to windowWidth by speed {
3.          draw the car at position x
4.          if x < (windowWidth /2)
5.              speed ← speed + 0.10
6.          otherwise
7.              speed ← speed – 0.10
        }
```

It is quite easy to adjust our previous code to accomplish this:

```
int         x, y;
float       speed;

void setup() {
  size(600,300);
  x = 0;
  y = 300;
  speed = 0;
}

void draw() {
  background(255,255,255);
  drawCar(x);
  x = int(x + speed);
  if (x+100 > width)
     noLoop();
  if (x < width/2)
     speed = speed + 0.10;
  else
     speed = speed - 0.10;
}

void drawCar(int x) {
  // Draw the body
  fill(150,150,150);  // gray
  rect(x, y-30, 100, 20);
  quad(x+20,y-30,x+30,y-45,x+55,y-45,x+70,y-30);

  // Draw the wheels
  fill(0,0,0);  // black
  ellipse(x+20, y-10, 20, 20);
  ellipse(x+75, y-10, 20, 20);
  fill(255,255,255); // white
  ellipse(x+20, y-10, 10, 10);
  ellipse(x+75, y-10, 10, 10);
}
```

If we were to run the code, it seems as though the car speeds up, then slows down, but then it seems to crash (i.e., stop abruptly) at the end of the window. That is because we are using the back bumper of the car (i.e., position **x**) to decide when to speed up or slow down. Since we split the window ½ way for accelerating and decelerating, then the car will only reach a speed of 0 again when the back bumper reaches the end of the window. However, we stop the car before that (i.e., when the front bumper reaches the window's end).

To fix this, we simply need to adjust the **if** statement to accelerate/decelerate based on the center of the car as follows:

```
if ((x+50) < width/2)
   speed = speed + 0.10;
else
   speed = speed - 0.10;
```

If you make the change and run the code again, you will notice that the car oscillates back and forth now! Do you know why ?

Well, if you were to print out the **speed** value, you would notice that it goes from **0** to **7.5** and then back down to **0** … but then the value continues going backwards to **-6.6**!! So, the speed becomes negative. When we add this negative speed to the **x** value, it reduces the **x** value, making the car move backwards. So, how do we fix it ?

We just need to ensure that the speed never becomes negative. There are a couple of ways to do this … by adding one of these to the end of the **draw()** procedure:

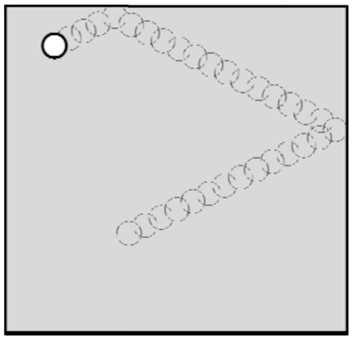
```
if (speed < 0)
    speed = 0;
```

```
speed = max(0, speed);
```

Example:

Now what about 2-dimensional motion ? How could we get a ball to bounce around the window so that it remains within the window borders ? To do this, we must understand the computational model.

To keep things simpler, lets assume that the ball is moving at a constant speed at all times. As the ball moves, we know that both its **x** and **y** locations will change. Also, the direction that the ball is facing should change. But when does the ball's direction change ? We will assume that it only changes direction when it hits the window borders.



So, we will need to keep track of the **ball's (x,y)** *location* as well as the *direction* (i.e., **angle**).

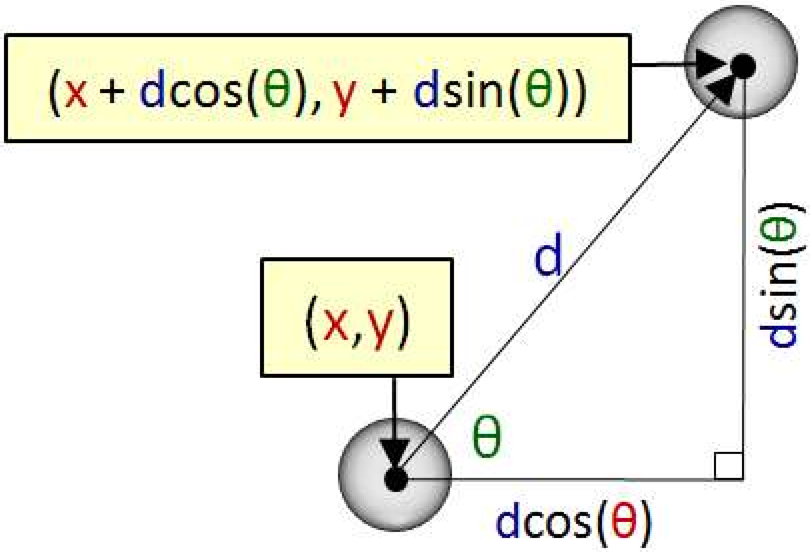
As with our moving car, we simply need to keep updating the ball's location and check to see whether or not it reaches the window borders. Here is the basic idea:

Algorithm: BouncingBall

```
1.      (x, y) ← center of the window
2.      direction ← a random angle from 0 to 2π
3.      repeat {
4.          draw the ball at position (x, y)
5.          move ball forward in its current direction
6.          if ((x, y) is beyond the window border) then
7.              change the direction accordingly
        }
```

It seems fairly straight forward, but two questions arise:

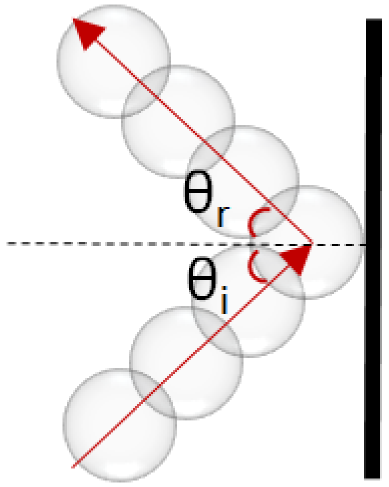
1) How do we "*move the ball forward in its current direction*" ?
2) How do we "*change the direction accordingly*" ?



The first is relatively simple, since it is just based on the trigonometry that we discussed it in the previous chapter. Given that the ball at location **(x,y)** travels distance **d** in direction **θ**, the ball moves an amount of **d•*cos*(θ)** horizontally and **d•*sin*(θ)** vertically as shown in the diagram. So, to get the new location, we simply add the horizontal component to x and the vertical component to y to get **(x + d*cos*(θ) , y + d*sin*(θ))**. Line 5 in the above algorithm therefore can be replaced by this more specific code (assuming that the ball moves at a speed of 10 pixels per iteration):

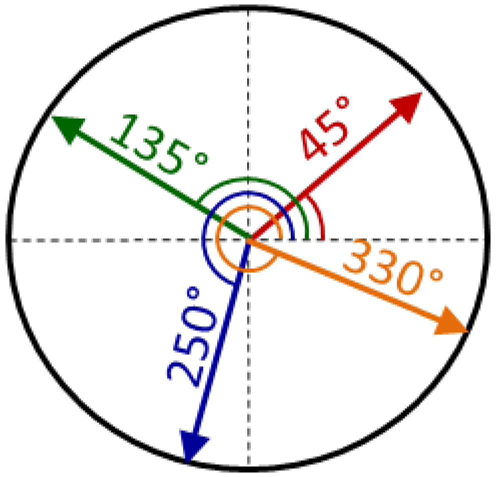
```
x ← x + 10 * cos(direction)
y ← y + 10 * sin(direction)
```

Now what about changing the direction when the ball encounters a window "wall" ? Well, we would probably like to simulate a realistic collision. To do this, we must understand what happens to a real ball when it hits a wall.



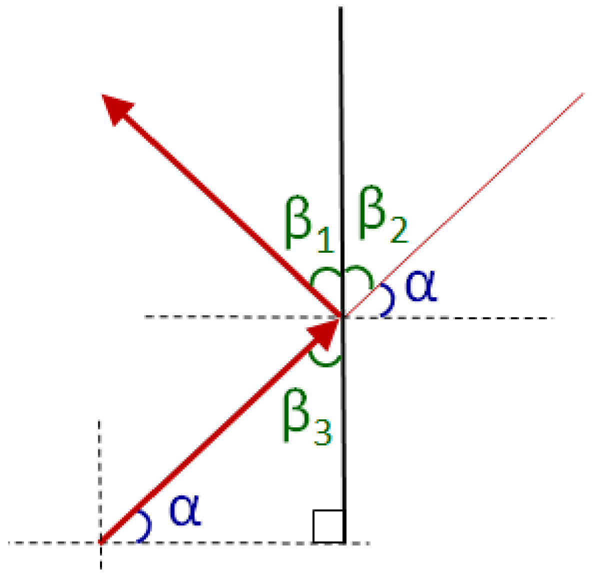
You may recall the ***law of reflection*** from science/physics class. It is often used to explain how light reflects off of a mirror. The law states that the ***angle of reflection*** is the same as the ***angle of incidence***, under ideal conditions. That is, the angle at which the ball bounces off the wall (i.e., θ_r in the diagram), will be the same as the angle at which it hit the wall (i.e., θ_i in the diagram).

However, where do we get the angle of incidence from ? Well, we have the direction of the ball stored in our **direction** variable.



This direction will always be an angle from **0** to **360°** (or from **0** to **2π** radians).

So, our ball's direction (called **α** for the purpose of this discussion) is always defined with respect to **0°** being the horizontal vector facing to the right. **360°** is the same as **0°**. As the direction changes counter-clockwise, the angle will increase. If the direction changes clockwise, the angle decreases. It is also possible that an angle can become negative. This is ok, since **330°** is the same as **-30°**.



Now, if you think back to the various angle theorems that you encountered in your math courses, you may remember these two:

1) the opposite angles of two straight crossing lines are equal

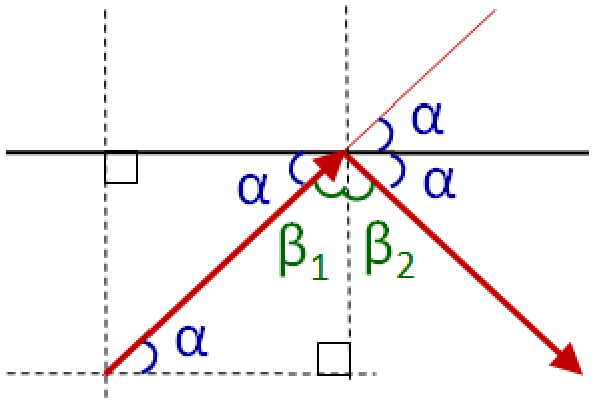
2) the interior angles of a triangle add up to **180°**

So, in the diagram on the right, for example, the 1st theorem above tells us that opposite angles $\boldsymbol{\beta_2}$ and $\boldsymbol{\beta_3}$ are equal. From the law of reflection, we also know that $\boldsymbol{\beta_1}$ and $\boldsymbol{\beta_3}$ are equal. Finally, **α** and $\boldsymbol{\beta_3}$ add up to **90°**.

What does all this mean ? Well, since α is the ball's direction, then to reflect off the wall, we simply need to add $\boldsymbol{\beta_1}$ and $\boldsymbol{\beta_2}$ to rotate the direction counter-clockwise. And since $\boldsymbol{\beta_1}$, $\boldsymbol{\beta_2}$ and $\boldsymbol{\beta_3}$ are all equal … and equal to **90° - α**, then to have the ball reflect we just need to do this:

direction = **direction** + (β_1 + β_2)
= **direction** + (90° - α + 90° - α)
= **direction** + (180° - 2 x **direction**)
= 180° - **direction**

The vertical bounce reflection is similar. In the diagram here, it is easy to see that $\boldsymbol{\beta_1} = \mathbf{90° - \alpha}$. To adjust for the collision on the top of the window, we simply need to subtract **2α** from the direction:

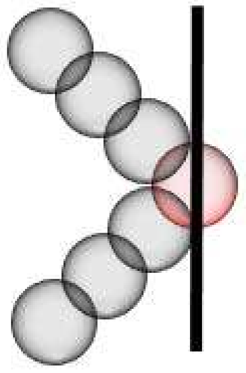


direction = **direction** - 2 x **direction**)
= **- direction**

To summarize then, when the ball reaches the left or right boundaries of the window, we negate the direction and add **180°**, but when it reaches the top or bottom boundaries, we just negate the direction. Here is how we do it:

```
Algorithm: BouncingBall
        windowWidth, windowHeight:        dimensions of the window

1.      x ← windowWidth/2
2.      y ← windowHeight/2
3.      direction ← a random angle from 0 to 2π
4.      repeat {
5.          draw the ball at position (x, y)
6.          x ← x + 10 * cos(direction)
7.          y ← y + 10 * sin(direction)
8.          if ((x >= windowWidth) OR (x <= 0)) then
9.              direction = 180° - direction
10.         if ((y >= windowHeight) OR (y <= 0)) then
11.             direction = - direction
        }
```

Our calculations made the assumption that the window boundaries are horizontal and vertical. Similar (yet more complex) formulas can be used for the case where the ball bounces off walls that are placed at some arbitrary angle. Also, all of our calculations assumed that the ball was a point. In reality though, the ball has a shape. If, for example, the ball was drawn as a circle centered at **(x,y)**, then it would only detect a collision when the center of the ball reached the border.



How could we fix this ?

We just need to account for the ball's radius during our collision checks:

```
if (((x+radius) >= windowWidth) OR (x-radius)  <= 0)) then
        …
if ((y+radius)  >= windowHeight) OR (y-radius)  <= 0)) then
        …
```

In processing, the code follows directly from, the pseudocode:

```
int             x, y;           // location of the ball at any time
float           direction;      // direction of the ball at any time

static float    SPEED = 10;     // the ball's speed
static int      RADIUS = 15;    // the ball's radius

void setup() {
  size(600,600);
  x = width/2;
  y = height/2;
  direction = random(TWO_PI);
}

void draw() {
  // erase the last ball position and draw the ball again
  background(0,0,0);
  ellipse(x, y, 2*RADIUS,2*RADIUS);

  // move the ball forward
  x = x + int(SPEED*cos(direction));
  y = y + int(SPEED*sin(direction));

  // check if ball collides with borders and adjust accordingly
  if ((x+RADIUS >= width) || (x-RADIUS <= 0))
    direction = PI - direction;
  if ((y+RADIUS >= height) || (y-RADIUS <= 0))
    direction = -direction;
}
```

Notice that the angles are in radians, instead of degrees. That is because the trigonometric functions **cos()** and **sin()** require angles in radians. Just for reference, **TWO_PI** and **PI** are constants defined in Processing that represent **2π** (i.e., **0°** or **360°**) and **π** (i.e., **180°**)

3.3 Event Handling

In many simulations, especially games, it is important to interact with the user. Sometimes the interaction is in regards to setting various simulation parameters before the simulation begins (i.e., wind speed, sensing diameter, start/end times, etc.). For gaming, there are other appropriate parameters such as difficulty settings, level of detail, game level, etc..

However, it is often necessary to interact with the user during the simulation by the means of buttons being pressed on the window, data being entered through dialog boxes, mouse clicks and keyboard presses, etc..

A problem arises, however, when dealing with such events. The problem is that the program is usually busy processing and displaying data in an endless loop. Since typical computers have only one processor (ignore multi-core for now), they can only literally do one thing at a time. However, the operating system of the computer is set up to handle user interaction ***events*** that arise while the computer is running.

> *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.*

These are called low-level events because they deal directly with physical interaction with the user. There are higher-level events that differentiate what the user is actually tying to do. For example, in JAVA, there are window-related events that get generated when the user clicks on a button, enters text in a text field or selects something from a list, etc. Regardless of the type of event that occurs, the programmer can decide what to do when these events occur by writing event handling routines:

> *An **event handler** is a procedure that specifies the code to be executed when a specific type of event occurs in the program.*

Typically, when dealing with user-interaction in an application, a programmer will write many event handlers, each corresponding to unique types of events. In fact, the event handlers may have different code to evaluate depending on the context of the program. For example, in a game, a mouse click may cause the game character to shoot a weapon when in one “mode” of the game, but when in another mode, the same mouse click may simply allow objects to be selected and moved around.

Some programs rely solely on events to determine what they will do. These are known as ***event-driven programs***. After initializing some values and perhaps opening a window to display something, event-driven programs enter into an infinite loop, called an ***event loop***.

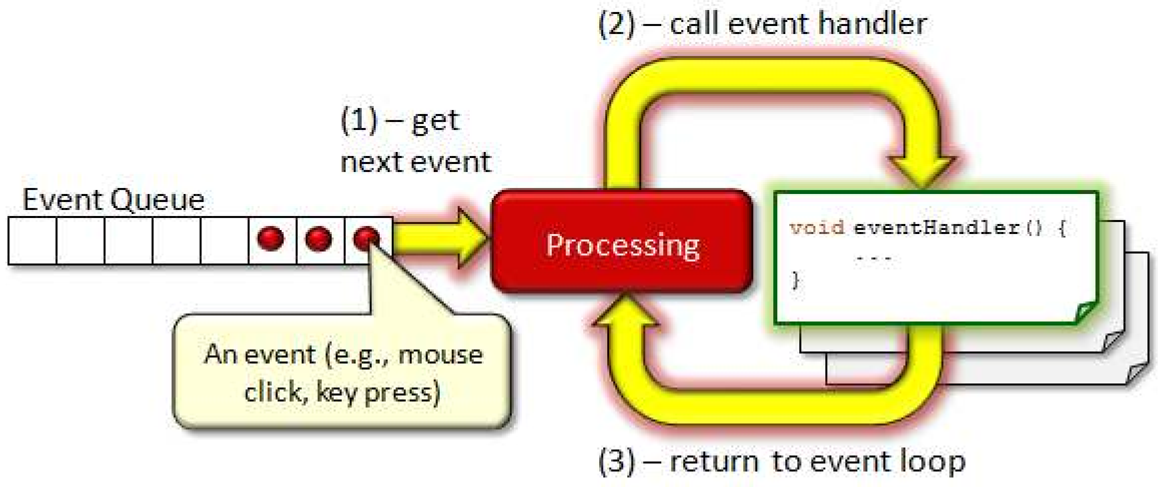
*An **event loop** is an infinite loop that waits for events to occur. When an event arrives, it is handled and then the program returns to the loop to wait for the next event.*

The idea of an event loop is similar to the notion of a store clerk waiting for customers ... the clerk does nothing unless an "event" occurs. Here are some events which may occur, along with how they may be handled:

- a customer arrives – store clerk wakes up and looks sharp
- a customer asks a question - store clerk gives an answer
- a customer goes to the cash to buy - store clerk handles the sale
- time becomes 6:00pm - store clerk goes home

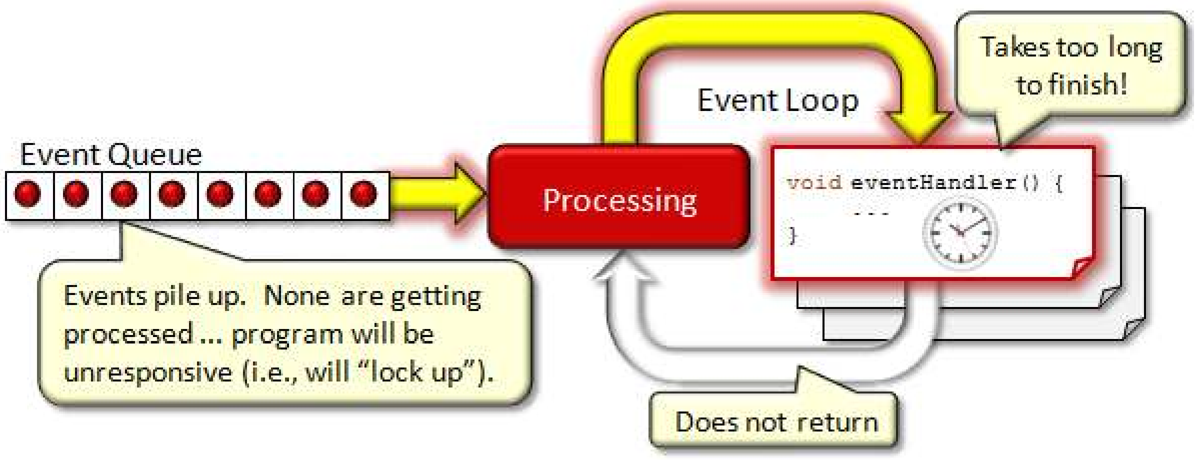
When multiple customers arrive at the same time, only one can be served, so the others wait in line. Once a customer has been served, the next customer in line is served. This repeats until there are no more customers, in which case the store clerk waits patiently again for more customers or "events".

This line-up of customers is similar to what happens with the event loop. There is an ***event queue*** which is a "line-up" of incoming events that need to be handled (e.g., mouse click, key press, mouse move, etc…). Once an event has been handled to completion, the event loop extracts the next event from the queue and handles it:



Events are handled one at a time on a first-come-first-served basis. Event-driven programs continually operate in this manner. Often, the event queue is empty. In a typical application, the time between mouse clicks and keyboard presses is so large, that it is rare to have more than one or two events in the queue at any time. However, when a timer is used, it may generate events very quickly (e.g., once per millisecond) and the event queue can fill up quickly if the event handler is too slow.

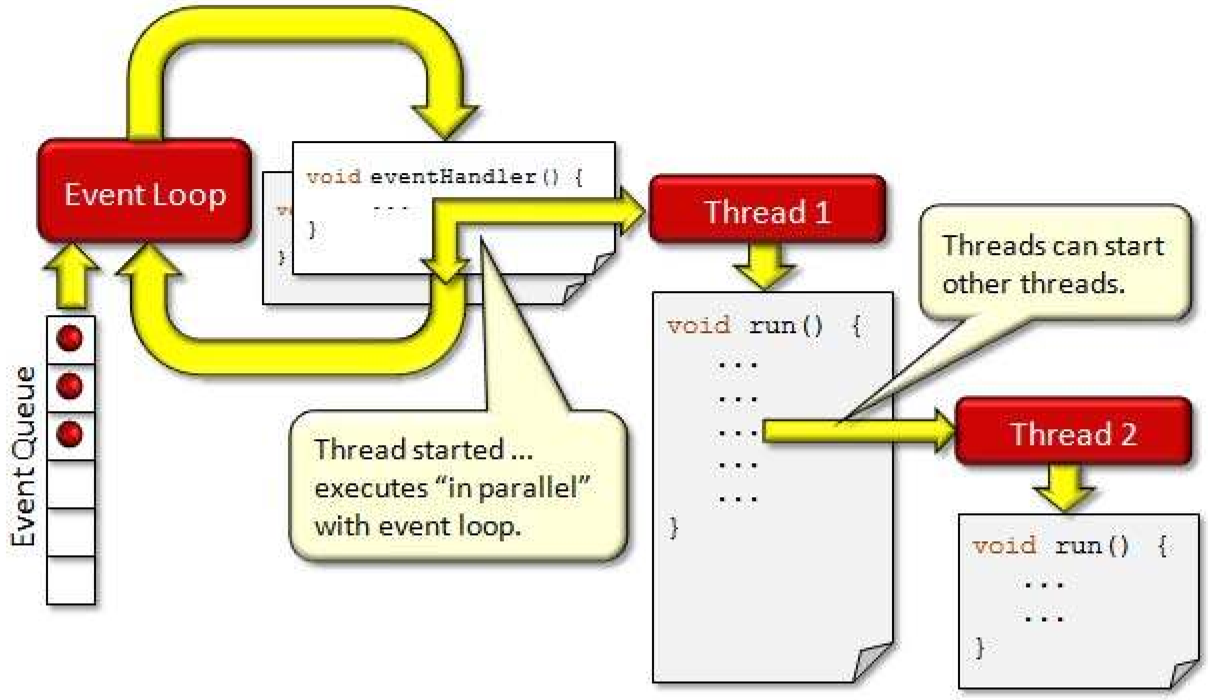
For example, assume that your program sent an email when you clicked on a "Send" button. The clicking on the button is an event which will call an event-handling procedure. Assume that the event handler attempts to send the email. If it takes too long (as it usually does) to send the email then the event handler will not return right way. Meanwhile, the user may be trying to click on various places on the window … but with no response. The window itself cannot be closed. What is happening is that when the user clicks, moves the mouse, presses a key etc.., these events are simply placed into the queue. However, the event loop is not taking any new events from the queue since it has not returned from the previous event. So the system has locked up until the "send email" event handler has returned.


Event Queue
Event Loop
Takes too long to finish!
Processing
void eventHandler() {
...
}
Events pile up. None are getting processed ... program will be unresponsive (i.e., will "lock up").
Does not return

So, it is important to make sure that event handling procedures do not take up too much time. If there is no choice in that the event handler may take a long time, then we can get around this "program lock-up" by spawning (i.e., starting up) what is called a ***thread***.

*A **thread (**a.k.a. **process)** is a unit of processing that performs some tasks. It generally results from a fork (i.e., split) of a program into two or more tasks that run at the same time.*

The diagram below describes how a thread is started:



As shown in the diagram, a thread (i.e., new task) is started, perhaps from an event handling routine or from another thread. This thread acts just like a separate program (although it shares the same variable space as the program that it started from (i.e., was spawned from). The new thread continues on evaluating program code concurrently (i.e., at the same time as) the code that spawned it. So, we could, for example, spawn a thread to go send that email which takes a long time, while the original program continues on back to the event loop to handle more events. The newly spawned thread may simply stop when it has completed, or it too may loop repeatedly, performing other program-related tasks .. and it too can spawn additional threads at any time.

Although creating and starting threads is easy, writing programs that have multiple inter-communicating threads can be difficult. There are many timing-related issues that arise as well as the area of resource management. Programming with many threads/processes falls into the computer science areas of concurrent programming and parallel & distributed computing. We will not discuss this any further in the course.

3.4 User Interaction in Processing

Recall that in processing, the main event-handling loop is hidden. However, it allows you to write your own event handling procedures for some pre-defined types of events. Also, processing has a kind of “internal event handler” that sets the values of some useful variables which are related to the mouse and keyboard events. Here is a table of the mouse-related variables that are automatically set:

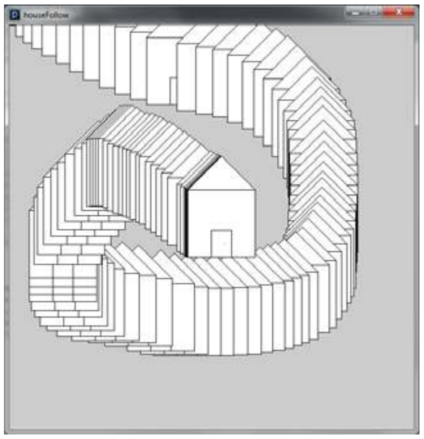
Variable	Description
mouseX	The current **x** (i.e., horizontal) position of the mouse
mouseY	The current **y** (i.e., vertical) position of the mouse
pmouseX	The previous **x** (i.e., horizontal) position of the mouse
pmouseY	The previous **y** (i.e., vertical) position of the mouse
mousePressed	**true** if the mouse button is being pressed, otherwise **false**
mouseButton	The mouse button that is being pressed (i.e., always one of **LEFT**, **RIGHT** or **CENTER**).

We can make use of these variables if we want our program to interact with the user based on the mouse movements and keyboard keys that are pressed.

Example:

The following code will draw a house with its bottom-left at the mouse’s current location:

```
void setup() {
     size(600,600);
}

void draw() {
     drawHouse(mouseX, mouseY);
}

void drawHouse(int x, int y) {
     rect(x, y-100,100,100);
     triangle(x,y-100,(x+50),y-150,(x+100),y-100);
     rect(x+35,y-40,30,40);
     point(x+55,y-20);
}
```



Notice how the house is draw repeatedly at whatever position the mouse is at currently. How could we alter this code so that it erases the old house ? Here are 2 choices:

a) we can erase the background each time or
b) erase the house at its previous location before drawing the new one.

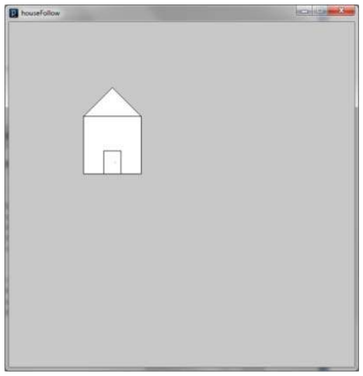
Here is solution a):

```
void draw() {
  background(200);    // light gray = 200
  drawHouse(mouseX, mouseY);
}
```

We just needed to add one line at the top of the **draw()** procedure that repaints the background … in this case gray with level 200 … a light gray.

Here is solution b):

```
void setup() {
     size(600,600);
     background(200);
}

void draw() {
     fill(200);
     stroke(200);
     drawHouse(pmouseX, pmouseY);
     fill(255);
     stroke(0);
     drawHouse(mouseX, mouseY);
}

void drawHouse(int x, int y) {} // same code as above… omitted to save space
```



Notice how it makes use of the previous mouse position (`pmouseX, pmouseY`) and "undraws" the house at that previous location. Some programming languages do not have such convenient variables. For example, if **pmouseX** and **pmouseY** were not available, how would you accomplish the same thing ? We would need to keep track of the previous position ourselves:

```
int  previousX, previousY;

void setup() {
    size(600,600);
    background(200);
    previousX = mouseX;
    previousY = mouseY;
}

void draw() {
    fill(200);
    stroke(200);
    drawHouse(previousX, previousY);
    fill(255);
    stroke(0);
    drawHouse(mouseX, mouseY);
    previousX = mouseX;
    previousY = mouseY;
}
```

Example:

How could we adjust the above code so that when the left mouse button is pressed, the house shrinks in scale but when the right mouse button is pressed, the house grows in scale ?

Well, to begin, we would need to recall our code that draws a house at a specific scale:

```
float scale = 1;

void drawHouse(int x, int y) {
  rect(x, y-100*scale,100*scale,100*scale);
  triangle(x,y-100*scale,(x+50*scale),
           y-150*scale,(x+100*scale),y-100*scale);
  rect((x+35*scale),y-40*scale,30*scale,40*scale);
  point((x+55*scale),y-20*scale);
}
```

We can then combine this code with some logic that increases or decreases the scale depending on the mouse button that was pressed. Here is the logic that will increase/decrease the scale by 1% each time:

```
if (left mouse button is being pressed) then
        scale ← scale * 0.99
if (right mouse button is being pressed) then
        scale ← scale * 1.01
```

However, in processing, the determination of whether or not a mouse button is being pressed is a separate function from the one that identifies which mouse button is being pressed. So, we cannot tell if the “left mouse button is being pressed” with one function call. First, we must determine whether or not a mouse button is being pressed by checking the **mousePressed** boolean variable. Then we may check which button is being pressed by checking the **mouseButton** variable as follows:

```
float scale;

void setup() {
    size(600,600);
    background(200);
    scale = 1;
}

void draw() {
    background(200);
    drawHouse(mouseX, mouseY);
    if (mousePressed) {
        if (mouseButton == RIGHT)
           scale = scale * 1.01;
        else
           scale = scale * 0.99;
    }
}

void drawHouse(int x, int y) {
  rect(x, y-100*scale,100*scale,100*scale);
  triangle(x,y-100*scale,(x+50*scale),
           y-150*scale,(x+100*scale),y-100*scale);
  rect((x+35*scale),y-40*scale,30*scale,40*scale);
  point((x+55*scale),y-20*scale);
}
```

Similarly, here is a table of the keyboard-related variables that are automatically set:

Variable	Description
keyPressed	**true** if a keyboard key is being pressed, otherwise **false**
key	A character representing the key that was most recently pressed (e.g., '**a**', '**A**', '**1**', '**.**', etc.., **BACKSPACE**, **TAB**, **ENTER**, **RETURN**, **ESC**, and **DELETE**).
keyCode	A constant representing a "special" key that was most recently pressed (i.e., **UP**, **DOWN**, **LEFT**, **RIGHT** arrow keys and **ALT**, **CONTROL**, **SHIFT**).

#

Example:

How would we adjust our above example code so that the house becomes darker when the **up arrow** key is pressed and lighter when the **down arrow** is pressed ?

We would need to add a **shade** variable to keep track of the current color of the house and adjust it accordingly:

```
shade ← 128
if (up arrow is being pressed) then
        shade ← shade + 1
if (down arrow is being pressed) then
        shade ← shade – 1
```

Of course, we would need to ensure that our **shade** always remains in the range of **0** to **255**:

```
if (up arrow is being pressed) AND (shade < 255) then
        shade ← shade + 1
if (down arrow is being pressed) AND (shade > 0)  then
        shade ← shade – 1
```

In processing, the code follows from this logic. We would first need to check to make sure that a key is being pressed using the **keyPressed** variable, and then (since the arrow keys are special keys) use **keyCode** to see whether or not it was the up or down arrow.

Here is the code:

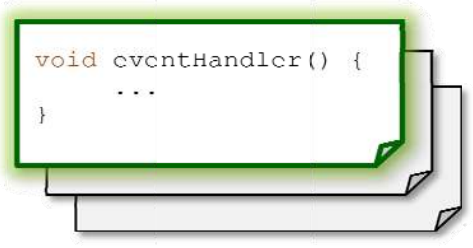
```
float scale;
int   shade;

void setup() {
    // same code as before ... omitted to save space
    shade = 128;
}

void draw() {
    background(200);
    fill(shade);
    drawHouse(mouseX, mouseY);
    if (mousePressed) {}  // same code as before ... omitted to save space
    if (keyPressed) {
        if (key == CODED) { // Required before checking “special” keys
            if ((keyCode == UP) && (shade < 255))
                shade = shade + 1;
            else if ((keyCode == DOWN) && (shade > 0))
                shade = shade - 1;
        }
    }
}
void drawHouse(int x, int y) {} // same as before ... omitted to save space
```

You may have noticed that we checked to see whether or not the **key == CODED**. This is required whenever checking for special keys. If, for example we were checking for keys **‘a’** and **‘b’**, instead of **UP** and **DOWN**, the code would simply use the **key** variable as follows:

```
if (keyPressed) {
    if ((key == 'a') && (shade < 255))
        shade++;
    else if ((key == 'b') && (shade > 0))
        shade--;
}
```

The above examples show how to use some of the pre-defined variables in Processing. However, not all programming languages have such variables readily available. In JAVA, for example, if you want to access the mouse position or determine the key that was pressed,, it is necessary to write your own event handler.

Processing allows you to write event handlers for some predefined types of events. In order to have this event handler called automatically when the event occurs, it is important to “spell” the event handler exactly the way that Processing is expecting.



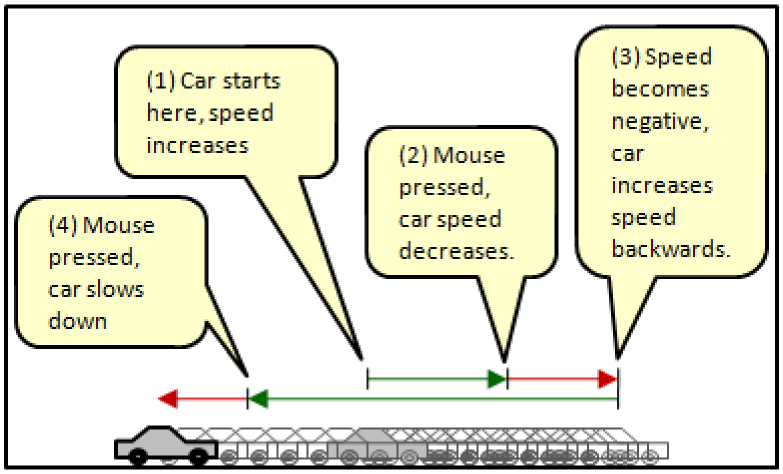
Below is a table showing which procedures need to written in order to handle some specific events:

Event Handling Procedure	**Description**
void mousePressed() { … }	called when a mouse button is pressed
void mouseReleased() { … }	called when a mouse button is released
void mouseClicked() { … }	called when a mouse button is pressed & released
void mouseMoved() { … }	called when the mouse is moved
void mouseDragged() { … }	called when the mouse is moved while a button is being pressed
void keyPressed() { … }	called when a key is pressed on the keyboard
void keyReleased() { … }	called when a key is released on the keyboard
void keyTyped() { … }	called when a key is pressed and released on the keyboard

Example:

Can we adjust the car program so that it accelerates horizontally towards the right but then when we press a mouse button, it should slow down and change directions, heading backwards to the left. Again, if we press a mouse button it should slow down and head right again. Therefore the car alternates from right to left all the while remaining within the window.

This is a good example of where an event handler could be used. The car should speed up at all times, only slowing down on a direction change (indicated by a mouse press event).


(1) Car starts here, speed increases
(2) Mouse pressed, car speed decreases.
(3) Speed becomes negative, car increases speed backwards.
(4) Mouse pressed, car slows down

As usual, we need to understand the model first. Below is the sequence of speed values (assuming the speed starts at 0 and accelerating/decelerating with a value of 1) that would occur as the car is moving. Notice how the speed changes when the event occurs (note that this goes a bit beyond the car movement shown above):

mouse click						🟥										🟥										
speed	0	1	2	3	4	5	4	3	2	1	0	-1	-2	-3	-4	-5	-4	-3	-2	-1	0	1	2	3	4	5
increment	1	1	1	1	1	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	1	1	1	1	1	1	1	1	1	1
car direction	→	→	→	→	→	→	→	→	→	→		←	←	←	←	←	←	←	←	←		→	→	→	→	→

Notice how the speed increases by the increment value until the mouse is clicked. It then "increases" by a negative increment value (i.e., decreases) until the next mouse click. We can make use of an **acceleration** variable that it is positive when speeding up and negative when slowing down, corresponding to the **increment** value in the above table.

Recall our algorithm that caused the car to speed up until the car reached half way, then slowed down:

```
Algorithm: AcceleratingCar
        windowWidth:            width of the window

1.    speed ← 0
2.    repeat {
3.        draw the car at position x
4.        x ← x + speed
5.        if x < (windowWidth /2) then
6.            speed ← speed + 0.10
7.        otherwise
8.            speed ← speed – 0.10
      }
```

Our new algorithm will be similar except that we now need to incorporate the acceleration/deceleration and also adjust to ensure that the car does not go beyond the window boundaries:

```
Algorithm: AlternatingCar
        windowWidth:            width of the window

1.    speed ← 0
2.    acceleration ← 0.10
3.    repeat {
4.        draw the car at position x
5.        x ← x + speed
6.        speed ← speed + acceleration
7.        if x < 0 then {
8.            speed ← 0
9.            x ← 0
          }
10.       if (x +100) > windowWidth then {
11.           speed ← 0
12.           x ← windowWidth - 100
          }
     }
```

All that remains is to set up an event handler for when the mouse button is clicked. In that event handler, all we need to do is negate the acceleration:

```
acceleration ← acceleration * -1
```

That is all. Here is the corresponding Processing code:

```
int        x, y;
float      speed;
float      acceleration;

void setup() {
  size(600,300);
  x = 0;
  y = 300;
  speed = 0;
  acceleration = 0.10;
}

void draw() {
  background(255,255,255);
  drawCar(x);

  x = int(x + speed);
  speed = speed + acceleration;

  if (x < 0) {
    x = 0;
    speed = 0;
  }
  else if (x+100 > width) {
    x = width - 100;
    speed = 0;
  }
}

// This is the mouse pressed event handler
void mousePressed() {
  acceleration = acceleration * -1;
}

void drawCar(int x) {
  // … same code as before … omitted to save space.
}
```

Example:

Recall our bouncing ball example. Below is the adjusted code to ensure that the ball remains within the screen and with a speed variable added:

```
Algorithm: BouncingBall
          windowWidth, windowHeight:        dimensions of the window
          radius:                           the ball's radius

1.        x ← windowWidth/2
2.        y ← windowHeight/2
3.        direction ← a random angle from 0 to 2π
4.        speed ← 10
5.        repeat {
6.              draw the ball at position (x, y)
7.              x ← x + speed * cos(direction)
8.              y ← y + speed * sin(direction)
9.              if ((x+radius >= windowWidth) OR (x-radius <= 0)) then
10.                 direction = 180° - direction
11.             if ((y+radius >= windowHeight) OR (y-radius <= 0)) then
12.                 direction = - direction
          }
```

How can we adjust this code using event handlers so that the ball can be "grabbed" and "thrown" by the mouse ? That is, if the user places the mouse cursor over the ball and clicks, then the ball stops moving and appears to be "stuck" to the mouse cursor until the mouse is released. Then when we let go of the mouse button, the ball should "fly off" in the direction that we threw it with a speed that varies according to how "hard" we threw it.

To do this, we should break the problem down into more manageable steps:

1. Add the ability to grab the ball and carry it around
2. Add the ability to throw the ball
3. Adjust the speed of the ball according to how "hard" we threw it.

To grab the ball, we would need to prevent lines 7 and 8 from being evaluated while the ball is being held. Instead, we would set the ball's location to be the mouse location.

We can create a boolean to determine whether or not the ball is being held:

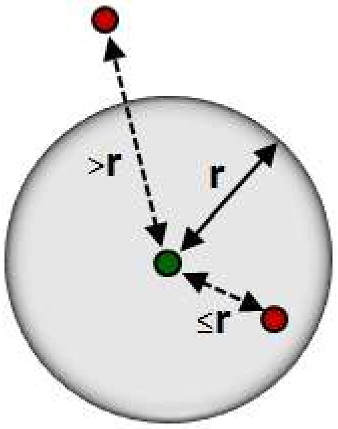
```
grabbed ← false
repeat {
    …
    if not grabbed then {
        x ← x + speed * cos(direction)
        y ← y + speed * sin(direction)
    }
    otherwise {
        x ← x position of mouse
        y ← y position of mouse
    }
    …
}
```

All that would be left to do is to set the **grabbed** variable accordingly. When the user would click on the ball, we should set it to **true** and when the user releases the mouse button, we should set it to **false**. So we need two event handlers:

```
mousePressed() {
    grabbed ← true
}
mouseReleased() {
    grabbed ← false
}
```

But this will ALWAYS "grab" the ball, even if the mouse cursor was not on it. How can we determine whether or not the mouse cursor is over the ball ? We can check to see whether or not the mouse location is within (i.e., **≤**) the ball's radius.

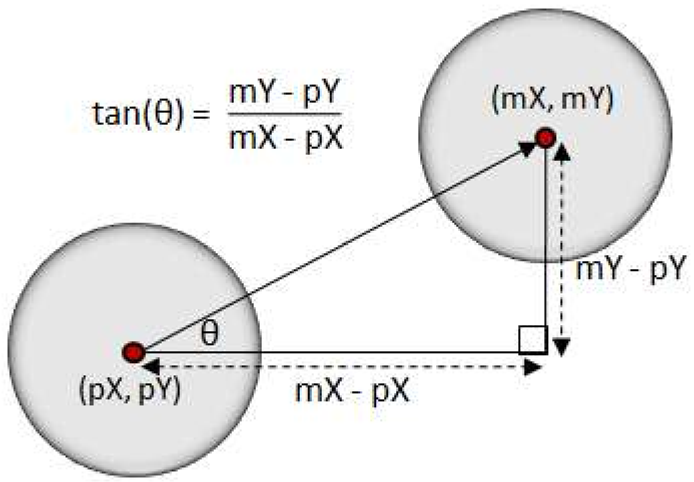
We can compute the distance from the ball's center (i.e., **(x,y)**) to the location of the mouse **(mX, mY)**. If this distance is less than or equal to the ball's radius, then we can assume that the user has "grabbed" the ball. Here is the adjusted code:



```
mousePressed() {
    d ← distance from (x, y) to (mX, mY)
    if (d <= radius) then
        grabbed ← true
}
```

When the user lets go of the ball, it will continue in the direction that it was in before it was grabbed. Now how do we adjust the code so that we are able to "throw" the ball in some particular direction ?

Well, upon releasing the mouse, we will need to determine which direction the ball was being thrown in and then set the **direction** variable directly. We can determine the direction that the ball was thrown in by examining the current mouse location **(mX, mY)** with respect to the previous mouse location **(pX, pY)**.



The angle (i.e., **θ**) at which the ball should be thrown will be the **arctangent** of the differences in **x** and **y** coordinates as shown here.

However, in the case that we throw vertically, the difference in **x** coordinates will be zero and we are not allowed to divide by zero. Fortunately, many computer languages have a function called **atan2(y, x)** which allows you to find the angle that a point makes with respect to the origin **(0,0)**. We can make use of this by assuming that **(pX,pY)** is the origin and translate **(mX,mY)** accordingly as follows: **atan2(mY-pY, mX-pX)**

So, upon a mouse release, we can do this:

```
mouseReleased() {
    if grabbed then {
        direction ← atan2(mY-pY, mX-pX)
        grabbed ← false
    }
}
```

Notice that we only change the direction when we have already grabbed the ball.

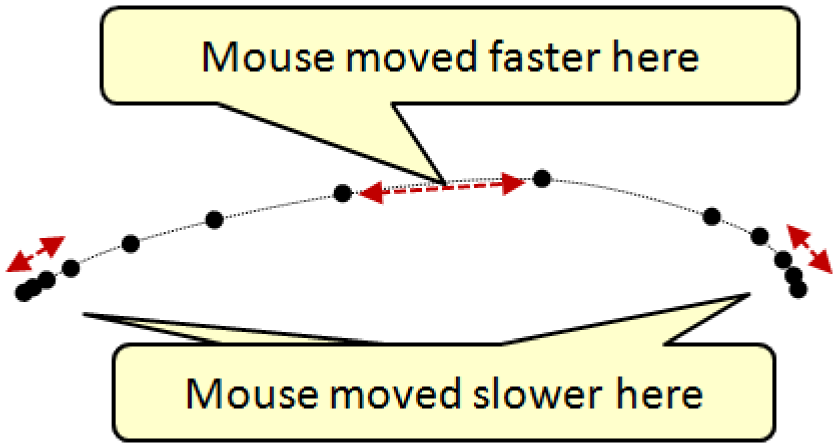
One last aspect of the program is to allow the ball to be thrown at various speeds. Likely, we want the ball to slow down as time goes on.

We should add the following to the algorithm's main **repeat** loop:

```
speed ← speed – 0.1
if (speed < 0) then
    speed ← 0
```

Now to determine the speed at which the ball is thrown, we can take notice of how the mouse location varies according to the speed at which it is moved.

If the mouse is moved fast, the successive locations of the mouse will be further apart, while slow mouse movements will have successive locations that are relatively closer together.



So, as a simple strategy, the amount of "throw hardness" can be computed as a function of the distance between the current mouse location and the previous mouse location.

We can simply set the **speed** to this distance in the **mouseReleased**() handler as follows:

```
mouseReleased() {
    if grabbed then {
        direction ← atan2(mY-pY, mX-pX)
        speed ← distance from (pX,pY) to (mX,mY)
        grabbed ← false
    }
}
```

Here is the resulting Processing code:

```
int         x, y;       // location of the ball at any time
float       direction;  // direction of the ball at any time
boolean     grabbed;    // true if the ball is being held
float       speed;      // the ball’s speed

final int   RADIUS = 40;             // the ball’s radius
final float ACCELERATION = 0.10;     // acceleration/deceleration amount

void setup() {
  size(600,600);
  x = width/2;
  y = height/2;
  direction = random(TWO_PI);
  grabbed = false;
  speed = 10;
}

void draw() {
  background(0,0,0);
  ellipse(x, y, 2*RADIUS,2*RADIUS);

  // move the ball forward if not being held
  if (!grabbed) {
    x = x + int(speed*cos(direction));
    y = y + int(speed*sin(direction));
  }
  else {
    x = mouseX;
    y = mouseY;
  }

  speed = max(0, speed - ACCELERATION);

  if ((x+RADIUS >= width) || (x-RADIUS <= 0))
    direction = PI - direction;
  if ((y+RADIUS >= height) || (y-RADIUS <= 0))
    direction = -direction;
}

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;
}
```

Chapter 4

Data Structures

What is in This Chapter ?

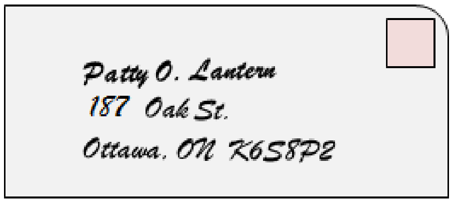
Almost all programs require the use of some data as input to the problem being solved. It is often advantageous to group (or structure) related data together. This chapter discusses the idea of creating ***data structures*** in Processing, which are also known as ***objects***. Objects are used as a way of keeping your data organized in a logical manner. In a later course, we will further develop the notion of an Object.

4.1 Data Structures and Objects

Recall that we used functions and procedures to provide ***control abstraction*** in order to hide low-level *conceptual details* within our algorithms so that they are simpler to read and understand. There is another type of abstraction known as ***data abstraction***. In this type of abstraction we are interested in hiding ***information*** *(or data)* that will unnecessarily clutter up an algorithm. The idea behind data abstraction is to group simple data values together which have a well understood relationship.

For example, if we are mailing out an envelope within the same country, then an ***address*** is assumed to have this information:

1. name
2. street number
3. street name
4. city
5. province
6. postal Code



Whenever most people hear the word **"address"**, they understand that such information is actually made up of some smaller, specific kinds of information. The address itself is not complete unless it is has all of that information. In a sense, the individual pieces of information *make up* (or ***define the structure of***) the address.

We can create such "more abstract" types of data (e.g., like an *address*) simply by combining or structuring the more primitives (i.e., simpler) pieces of data together in meaningful ways:

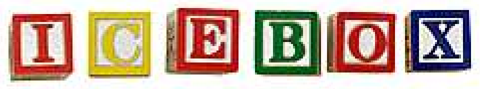
A ***data structure*** *is a particular way of combining, storing and organizing data so that it may be used more efficiently and in a more abstract manner.*

We also use the word ***data type*** which is somewhat analogous to the term ***data structure***. In object-oriented programming languages, such as Processing and Java, a data type is also known as a ***class*** or category, and *defining a data type* is also called ***defining an object***.

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

Recall that a primitive data type represents a **single** simple piece of information such as a number or character. 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 →

Perhaps the simplest data structure is called a ***string*** which is a group of one or more characters with a specific ordering. Characters by themselves are not very interesting. However, when we group them together, we form a huge variety of words and seemingly unlimited variety of sentences. Each word in the English language, for example, represents a string data structure, as does each sentence, paragraph, page of text, etc…



In many programming languages (including Processing and JAVA), strings are represented by placing double quotes around a set of characters like this:

name ← "Patty O. Lantern"

A string is not a *primitive* data type because it is made up of characters…which themselves are the primitive data types. In fact, we can abstract out the notion of a string even further by grouping strings together in a meaningful way to create an even more abstract data structure.

For example, consider an address as described above. A full address may be represented using multiple numbers and strings as follows:

name ← "Patty O. Lantern"
streetNumber ← 187
streetName ← "Oak St."
city ← "Ottawa"
province ← "ON"
postalCode ← "K6S8P2"

Together, all of the variables above represent a full address. It would be advantageous if we could define a single variable, perhaps called **address**, that can store all of the above information:

address ← … ? …

Of course, we could combine everything into one big string …

address ← "Patty O. Lantern, 187 Oak St., Ottawa, ON, K6S8P2"

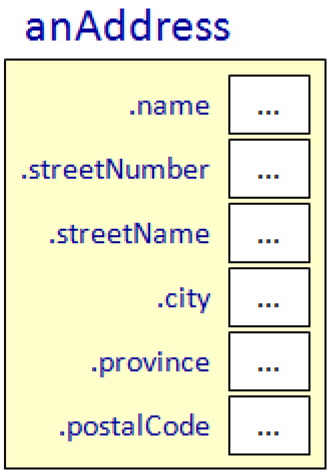
… but then it would be more difficult/cumbersome to extract the needed pieces of information (e.g., street number or last name).

Many programming languages allow you to "group" variables together into a structure of some type. The process of defining which variables and types of data should be grouped together is called ***defining a data structure*** (or ***defining a data type***). In object-oriented languages (such as JAVA) this is also called ***defining an object*** and sometimes ***defining a class***.

When defining such a structure, we need to specify the name of the new type of data (e.g., **Address**) as well as the names and types of data that is contained within it.

We will use the following notation to define a structure in our pseudocode (below), as well as visually (shown to the right):

```
define Address to be made of {
        .name
        .streetNumber
        .streetName
        .city
        .province
        .postalCode
}
```



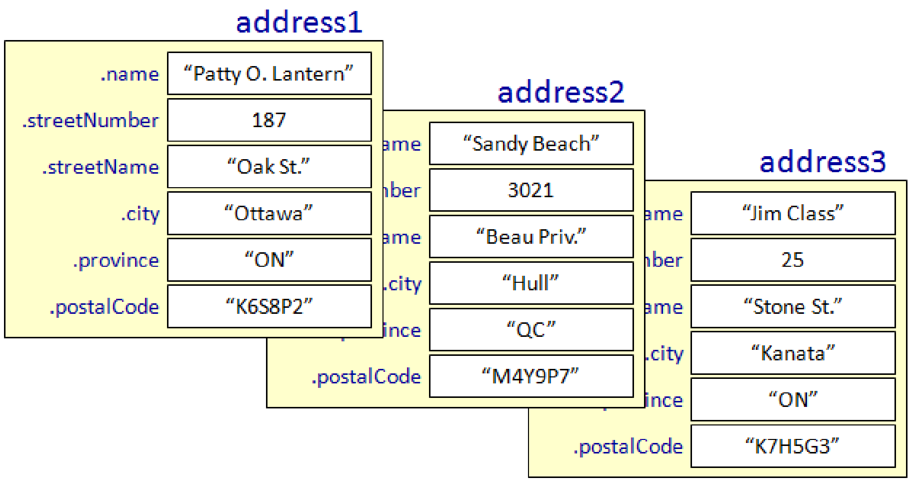
Notice that we capitalized the data type **A**ddress. This is proper coding style and any definition of data structure, data type or object should always be capitalized.

The above notation shows that an address is made up of 6 pieces of data with the given name labels. It is as if the **Address** data type is a "blank form" onto which we can fill in appropriate values. It defines a kind of *template* for creating data of this type.

That is how we will define a new data type. However, *defining* a data type does not actually create any variables, it only creates a *definition*. When we actually want to *use* a data type, we need a way of specifying that we want to create a new ***instance*** of this data type.

*An **instance** of a data structure (or object) is a particular group of values for each of the individual variables that make up the data structure (or object).*

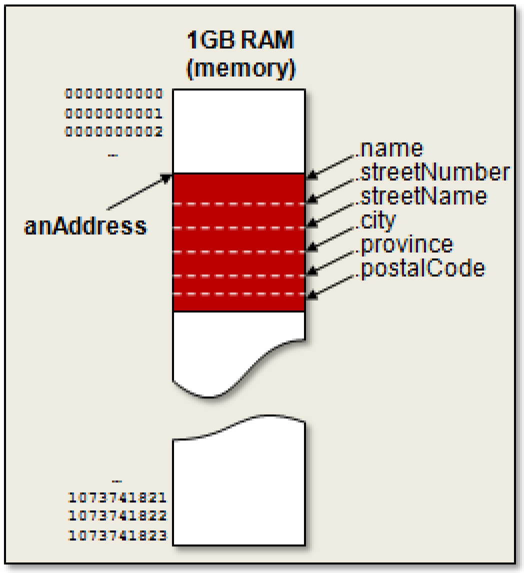
That means an instance is a particular object belonging to the category of objects defined by the data type. For example, each of the following is an instance of the **Address** data type, because they represent particular addresses:



Recall that when we created variables, we had to reserve space in the computer's memory that would allow us to store information in the variable. Similarly, in order to make a new variable that can hold a data type, we have to also declare the variable in order to allocate sufficient memory.

When we declare a variable that belongs to the new data type, however, it is as if we are declaring space for all of its pieces. In fact, the pieces of a particular instance of a data type are actually called ***instance variables***, since they are just regular variables … that happen to be grouped together to form a particular instance.

In some programming languages, you have to allocate/reserve space (i.e., memory) for the instance of the data structure yourself. The popular languages **C**, for example, requires you to call a function that will allocate the memory for you (e.g., **malloc**()). Once you are done with the variable, it is your responsibility to free up that memory space once again by calling another function (e.g., **free**()).



The allocating and freeing up of memory is known as ***dynamic memory allocation & deallocation*** and is quite tedious and unpleasant. It can also be a source of many problems/bugs/errors in a program. For example, if memory is allocated many times, but never freed, the program will fill up the computer's memory and the program will crash. These are called ***memory leaks***. Also, you have to make sure that you allocate enough memory to hold everything that you need and that you don't free memory that you still want to use … otherwise your program will likely crash.

In order to make programmers live happier lives, some language designers have decided that it would be convenient and safer to perform this ***memory management*** for you. That is, they provide a means of allowing you to automatically reserve memory when you want to make a new instance of a data type (i.e., when you want to use a new object in your program). Object-oriented languages such as C++, C#, JAVA and Processing have built-in memory management.

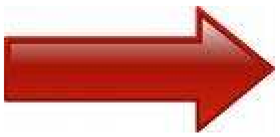
In order to allocate enough memory for a new instance of a data type and start using it, the **new** keyword is used. For example, here is how we will indicate that we want a new instance of the **Address** data type in our pseudocode:

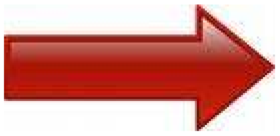
```
anAddress ← new Address
```

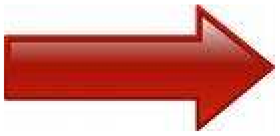
This will allocate enough memory to hold all of the address' information.

You may want to think of the **Address** class as a ***factory*** that makes **Address** objects (i.e., makes ***instances*** of the **Address** class). In general, every time we use **new**, it is as if we go to the factory for that class and buy a instance of the object. So… the *class* is the "factory", and the *instance* is the particular "object" that we can start using now in our programs.

The **Address** Class	**new Address**	Instance of type **Address**

		Patty O. Lantern 187 Oak St. Ottawa, ON K6S8P2

The **Person** Class	**new Person**	Instance of type **Person**
		

The **House** Class	**new House**	Instance of type **House**
		

We will then assign values to the individual components (i.e., instance variables) of **anAddress** by using the dot operator as follows:

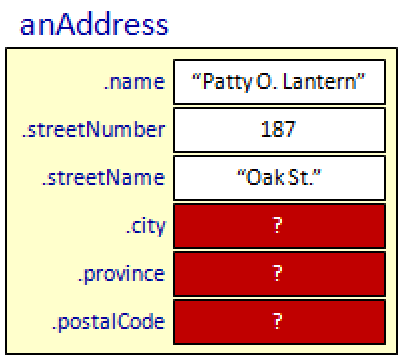
```
anAddress ← new Address
anAddress.name ← "Patty O. Lantern"
anAddress.streetNumber ← 187
anAddress.streetName ← "Oak St."
anAddress.city ← "Ottawa"
anAddress.province ← "ON"
anAddress.postalCode ← "K6S8P2"
```

The dot operator indicates that we are going inside the data type to get a piece of information. That is, we are getting more specific as to what particular piece of data we want. Whenever we use, for example, **anAddress.**name, it behaves just like any other variable and refers to the data stored in that part of the address's memory.

Sometimes, however, some information may be missing. For example, when we give a local person our ***address*** on a piece of paper, it is likely that we'll only give them the street number and name.

Patty O. Lantern
187 Oak St.

In this situation there will be some missing information. Perhaps the missing information is assumed to be particular values. For example, we may assume that the city and province are local to where we received the piece of paper. Nevertheless, when data is missing, we would want to make sure that we don't assume potentially wrong data. Our data structure, may therefore be missing data.



What is the value of the instance variables in this case, since we did not supply any values ? It actually depends what is in the computer's memory at the location where the city, province and postal code is stored. It could be "garbage" data that was from a data type whose memory was previously freed up. With memory-managed languages, however, these values are usually set to **0** (when the variable's type is a number) or **null** (when the variable's type is a data type).

> ***Null*** *is a word that represents an undefined value. If a variable has a value of null, it means that it does not yet have a value.*

Therefore, the following code logic is flawed:

```
anAddress ← new Address
print (anAddress.name)
```

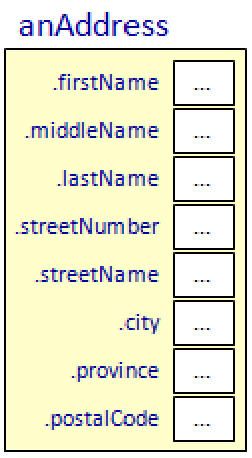
The code does not make sense because we are trying to print out the address's name before we have assigned a value to it. Depending on the language used, the result may be **null** or perhaps even random "garbage" data.

We can actually go deeper into each piece of data as well, making them more abstract. For example, we created the address's instance variable to store the **name** as a single string as follows:

```
anAddress.name ← "Patty O. Lantern"
```

It is sometimes desirable to be able to distinguish between the first name, last name and middle name(s) of the person. To do this, we would need to separate the names into different variables as follows:

```
anAddress.firstName ← "Patty"
anAddress.middleName ← "Ohh"
anAddress.lastName ← "Lantern"
```

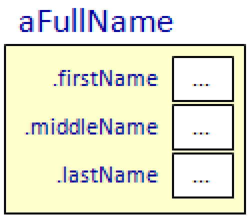


Then we can choose which portion of the person's name that we want to use at any time. A downside is that we now have to use 3 variables instead of 1.

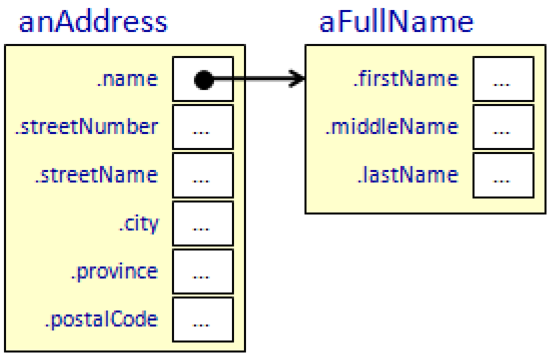
We could re-define the **Address** data type as shown in the picture here →

This would suffice. However, the address seems more complicated. We could perform additional data abstraction by combining the first, middle and last names into their own unique data structure:

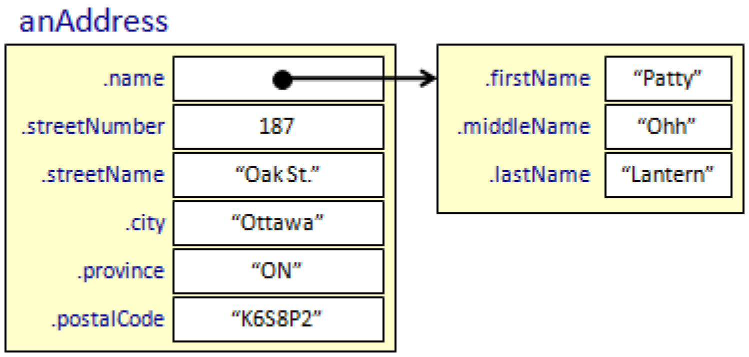
```
define FullName to be made of {
        .firstName
        .middleName
        .lastName
}
```



Then we could make use of this within our **Address** data structure as shown here. From the picture → you can see that the **name** stored in the **Address** structure is no longer a simple string of characters. Now it is a different kind of data structure (i.e., object) of type **FullName**.



Here is what our example would look like now:



The code for setting the name of **anAddress** would be as follows:

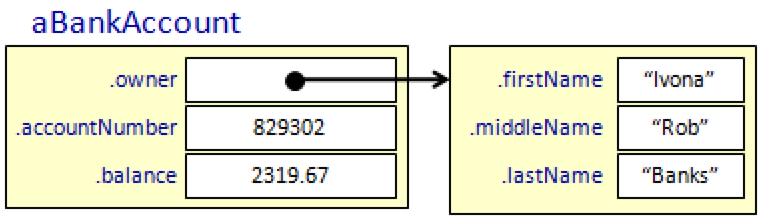
```
anAddress ← new Address
anAddress.name ← new FullName
anAddress.name.firstName ← "Patty"
anAddress.name.middleName ← "Ohh"
anAddress.name.lastName ← "Lantern"
```

Notice now that the dot operator is used twice: once to get into the address to get the name, then again to get into the name to set the first, middle and last values.

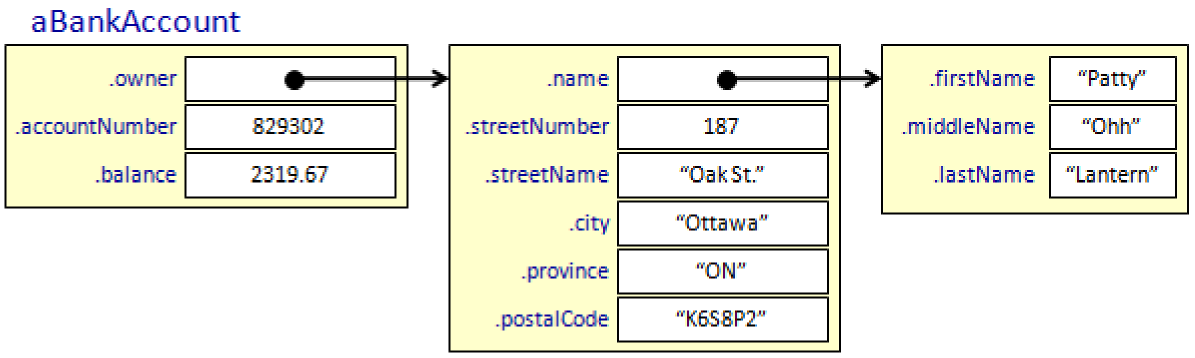
An additional advantage of creating the separate **FullName** data structure is that we can use it in other applications. Consider, a **BankAccount** data structure that is defined as follows:

```
define BankAccount to be made of {
        .owner
        .accountNumber
        .balance
}
```

We could then set the owner to be a **FullName** data type as well:



The fun does not stop here. In fact, it is often the case that data structures use multiple kinds of data structures within them. For example, could you determine the code that would produce this image ?



Now we should look at examples of using a data structure.

Example:

Recall the "Loan Qualification Kiosk" program that we implemented earlier which was to allow a people to enter personal information and then inform them as to whether or not they qualify for the loan.

```
1.  print welcome message
2.  employed ← ask user if he/she is currently employed
3.  hasDegree ← ask user if he received a univ. degree within past 6 months
4.  age ← ask user for his/her age
5.  yearsWorked ← ask user for # years worked at full time status

6.  if (employed is true) then {
7.      if (hasDegree is true) then print "Congratulations…"
8.      otherwise {
9.          if (age >= 30) then {
10.             if (yearsWorked >= 10) then print "Congratulations, …"
11.             otherwise print "Sorry, …"
            }
12.         otherwise print "Sorry, ...."
        }
    }
13. otherwise print "Sorry, ..."
```

How can we adjust the code to combine all of the employee information into one data structure ? Notice what the code looks like now when an **Employee** data structure is used:

```
define Employee to be made of {
        .employed
        .hasDegree
        .age
        .yearsWorked
}
```

Algorithm: LoanQualificationKiosk

```
1.      print welcome message
2.      employee ← new Employee
3.      employee.employed ← ask user if he/she is currently employed
4.      employee.hasDegree ← ask user if he received a univ. degree within past 6 months
5.      employee.age ← ask user for his/her age
6.      employee.yearsWorked ← ask user for # years worked at full time status

7.      if (employee.employed is true) then {
8.          if (employee.hasDegree is true) thenprint "Congratulations…"
9.          otherwise {
10.             if (employee.age >= 30) then {
11.                 if (employee.yearsWorked >= 10) then  print "Congratulations, …"
12.                 otherwise print "Sorry, …"
                }
13.             otherwise print "Sorry, ...."
            }
        }
14.     otherwise print "Sorry, ..."
```

All of the employee's information is packaged into the single **Employee** data structure and stored in the **employee** variable. The code seems longer, however, the data is now set up for more abstract use. For example, assume that we created a function to get the user's information and another to determine whether or not they qualify.

Notice the simple main code (i.e., lines 1 to 4):

Algorithm: LoanQualificationKiosk2

```
1.      print welcome message
2.      employee ← getUserInformation()
3.      determineQualifications(employee)
4.      print "Thank you, have a nice day."
```

Notice how the **Employee** object is created in the **getUserInformation()** procedure, populated with information from the user, and then returned to the main algorithm:

```
        getUserInformation() {
5.          print welcome message
6.          employee ← new Employee
7.          employee.employed ← ask if currently employed
8.          employee.hasDegree ← ask if received a univ. degree within past 6 months
9.          employee.age ← ask for age
10.         employee.yearsWorked ← ask for # years worked at full time status
11.         return employee
        }
```

The **determineQualifications()** procedure then accepts an incoming **Employee** object (which is labelled as **emp**) and uses it for various computations and decisions:

```
        determineQualifications(emp) {
12.         if (emp.employed is true) then {
13.             if (emp.hasDegree is true) then
14.                 print "Congratulations…"
                otherwise {
15.                 if (emp.age >= 30) then {
16.                     if (emp.yearsWorked >= 10) then
17.                             print "Congratulations, …"
18.                     otherwise print "Sorry, …"
                    }
19.                 otherwise print "Sorry, ...."
              }
            }
20.         otherwise print "Sorry, ..."
        }
```

Within the **determineQualifications()** procedure we simply use the dot operator to get at the specific piece of employee information that we need.

There are some advantages of using the data structure:

1) The main algorithm is more abstract and **simpler to understand**

2) If we add additional qualification parameters (e.g., marital status, # of dependants, credit history, etc…) then the **main program** (lines 1 through 4) **remains unchanged**.

The code is thus simpler and more organized with the use of the data structure/object.

However, it is not always obvious to know what kind of information (i.e., components) should *make up* a data structure/object. That is … there is not always a "well defined" set of data that make up the object. For an **Address** data structure, it is somewhat obvious. However, what about a **Person** data structure … what should "make up" a person ?

Some possible attributes of a person data structure may be **firstName**, **lastName**, **age**, **gender** and **retired**. Why would we choose these ? In reality, doesn't our choice of attributes depend on the application that we are trying to develop. For example, while the **age** and **gender** may be vital pieces of information for a program that determines players on a team sport in some league, information about whether a person is **retired** is not necessary. And for medical applications, perhaps **weight** and **height** are vital pieces of information. If it is to be an online social network application, perhaps **emailAddress** is an important piece of information that all Persons should have. The choice of a data structure's components really depends on the application.

As another example, consider defining a **Car** data structure. We should think of what characteristics we will need to store for each car (e.g., **make**, **model**, **color**, **mileage**, etc..):

The choice will depend on the program/application you are making. Consider these possible applications in which a **Car** data structure may be used:

- a program for a car repair shop
- a program for a car dealership
- a program for a car rental agency
- a program for an insurance company

So, now let us examine what kind of attributes (i.e., instance variables) that we would likely need to define for a **Car** in each of these individual applications:

- **repair shop**
  make, model, year, engine size, spark plug type, air/oil filter types, air hose diameter, repair history, owner etc..
  .
- **car dealership**
  model, price, warranty, interior finish (leather/material), color, engine size, fuel efficiency rating, etc...

- **rental agency**
  sedan or coupe, make, model, license plate, price per hour, mileage, repair history, etc...

- **insurance company**
  year, make, model, owner, insurance type (fire/theft/collision/liability), color, license plate, etc...

So you can see that it is not always straight forward to identify the components of a data structure. You need to always understand **how it fits** into the application.

Example:

Recall the algorithm that accelerates a car across the window:

```
speed ← 0
for x locations from 0 to windowWidth by speed {
    draw car at position (x, windowHeight –100)
    if x < (windowWidth /2) then
        speed ← speed + 0.10
    otherwise
        speed ← speed – 0.10
}
```

We can create a **Car** data structure and use it within the program. What should make up the data structure ? What components can be grouped together to represent the car ?

```
define Car to be made of {
    .x
    .y
    .speed
}
```

```
1.      myCar ← new Car
2.      myCar.x ← 0
3.      myCar.y ← windowHeight –100
4.      myCar.speed ← 0
5.
6.      for x locations from 0 to windowWidth by myCar.speed {
7.          myCar.x ← x
8.          draw(myCar)
9.          if myCar.x < (windowWidth /2) then
10.             myCar.speed ← myCar.speed + 0.10
11.         otherwise
12.             myCar.speed ← myCar.speed – 0.10
        }
```

Now all of the car's parameters (i.e., location and speed) are kept together. Notice as well that the **draw()** procedure now simply takes one parameter, representing the car. It can then extract the **x** and **y** locations easily with the dot operator.

This may not seem like an advantage in this simple example, but as more and more car-related functions or procedures are added, this will greatly reduce the number of parameters being passed around. Also, if we add additional instance variables to the car's structure (e.g., color, scale) then this information will be readily available in the **draw()** procedure and we will not have to change line **8** of our program !! Once again, the code is cleaner and more organized.

4.2 Using Objects in Processing/JAVA

In Processing (and Java), data structures are called ***objects***. To define a new data structure, we define a ***class*** using the **class** keyword and simply list the variables one after another (along with their types) between braces. Here are a few examples:

Pseudocode	**Processing code**
define FullName to be made of {	`class FullName {`
.firstName	`String     firstName;`
.middleName	`String     middleName;`
.lastName	`String     lastName;`
}	`}`
define Address to be made of {	`class Address {`
.name	`FullName   name;`
.streetNumber	`int        streetNumber;`
.streetName	`String     streetName;`
.city	`String     city;`
.province	`String     province;`
.postalCode	`String     postalCode;`
}	`}`
define BankAccount to be made of {	`class BankAccount {`
.owner	`Address    owner;`
.accountNumber	`int        accountNumber;`
.balance	`float      balance;`
}	`}`
define Employee to be made of {	`class Employee {`
.employed	`boolean    employed;`
.hasDegree	`boolean    hasDegree;`
.age	`int        age;`
.yearsWorked	`int        yearsWorked;`
}	`}`
define Car to be made of {	`class Car {`
.x	`int        x;`
.y	`int        y;`
.speed	`float      speed;`
}	`}`

Notice how the types must now all be specified. Also, notice that when one data structure is contained within another (e.g., **FullName** inside **Address**), we must indicate the name of the data structure (i.e., class) as the variable's type.

The above class definitions cannot be done within the **setup()** or **draw()** procedures, nor in any other procedures or functions. They are defined on their own, separately, just like the **setup()** and **draw()** procedures. It is a good idea to gather all your class definitions near the top or bottom of your program. Here is an example template showing **Car** and **House** data structures defined just above the **setup()** and **draw()** procedures:

```
int   aVariable;
float anotherVariable;

class Car {
     …
}

class House {
     …
}

void setup() {
     …
}

void draw() {
     …
}
```

Later, in COMP1406, when we start doing Java programming, we will define each class in its own separate file.

Now, to create a new *instance* of one of these classes, we usually need to first create a variable to store it in and then we simply use the **new** keyword followed by the class name and parenthesis. Then we assign values to the instance variables using the **=** operator. Here are some examples of how this is done:

Pseudocode	**Processing code**

myCar ← **new** Car
myCar.x← **0**
myCar.y← **windowHeight - 100**
myCar.speed← **0**
	`Car   myCar;`
	
	`myCar = new Car();`
	`myCar.x = 0;`
	`myCar.y = height – 100;`
	`myCar.speed = 0;`

aFullName ← **new** FullName
aFullName.firstName ← "Patty"
aFullName.middleName ← "Ohh"
aFullName.lastName ← "Lantern"
	`FullName   aFullName;`
	
	`aFullName = new FullName();`
	`aFullName.firstName = "Patty";`
	`aFullName.middleName = "Ohh";`
	`aFullName.lastName = "Lantern";`

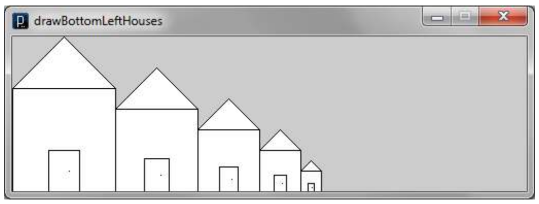
addr ← **new** Address
addr.name ← **new** FullName
addr.name.firstName ← "Patty"
addr.name.middleName ← "Ohh"
addr.name.lastName ← "Lantern"
addr.streetNumber ← 187
addr.streetName ← "Oak St."
addr.city ← "Ottawa"
addr.province ← "ON"
addr.postalCode ← "K6S8P2"
	`Address  addr;`
	
	`addr = new Address();`
	`addr.name = new FullName();`
	`addr.name.firstName = "Patty"`
	`addr.name.middleName = "Ohh"`
	`addr.name.lastName = "Lantern"`
	`addr.streetNumber = 187`
	`addr.streetName = "Oak St."`
	`addr.city = "Ottawa"`
	`addr.province = "ON"`
	`addr.postalCode = "K6S8P2"`

b ← **new** BankAccount
b.owner ← **new** Address
b.owner.name ← **new** FullName
b.owner.name.firstName ← "Patty"
b.owner.name.middleName ← "Ohh"
b.owner.name.lastName ← "Lantern"
b.owner.streetNumber ← 187
b.owner.streetName ← "Oak St."
b.owner.city ← "Ottawa"
b.owner.province ← "ON"
b.owner.postalCode ← "K6S8P2"
b.accountNumber ← 829302
b.postalCode ← 2319.67
	`BankAccount  b;`
	`Address      addr;`
	
	`addr = new Address();`
	`addr.name = new FullName();`
	`addr.name.firstName = "Patty"`
	`addr.name.middleName = "Ohh"`
	`addr.name.lastName = "Lantern"`
	`addr.streetNumber = 187`
	`addr.streetName = "Oak St."`
	`addr.city = "Ottawa"`
	`addr.province = "ON"`
	`addr.postalCode = "K6S8P2"`
	
	`b = new BankAccount();`
	`b.owner = addr;`
	`b.accountNumber = 829302`
	`b.postalCode = 2319.67`

emp ← **new** Employee
emp.employed ← **… some code …**
emp.hasDegree ← **… some code …**
emp.age ← **… some code …**
emp.yearsWorked ← **… some code …**
	`Employee  emp = new Employee();`
	
	`emp.employed = /* some code */;`
	`emp.hasDegree = /* some code */;`
	`emp.age = /* some code */;`
	`emp.yearsWorked = /* some code */;`

Example:

Recall our example in which we drew 5 houses of various sizes at adjacent locations on the window →



Here is the code that we used:

```
void setup() {
    size(500,150);

    drawHouse(0, 150, 1);
    drawHouse(100, 150, 0.8);
    drawHouse(180, 150, 0.6);
    drawHouse(240, 150, 0.4);
    drawHouse(280, 150, 0.2);
}

void drawHouse(int x, int y, float s) {
    rect(x, y-100*s,100*s,100*s);
    triangle(x,y-100*s,(x+50*s),y-150*s,(x+100*s),y-100*s);
    rect((x+35*s),y-40*s,30*s,40*s);
    point((x+55*s),y-20*s);
}
```

How could we adjust this code to make use of a House data structure ? What information in this example represents the attributes of the Houses ?

The x, y location are attributes of the houses as well as their scale factor. So, we could define a house data structure (i.e., object) as follows:

```
class House {
    int    x;
    int    y;
    float  s;
}
```

Then we can make use of this new **House** object in our code:

```
void setup() {
    House  h1, h2, h3, h4, h5;

    size(500,150);

    h1 = new House();
    h1.x = 0;
    h1.y = 150;
    h1.s = 1.0;

    h2 = new House();
    h2.x = 100;
    h2.y = 150;
    h2.s = 0.8;

    h3 = new House();
    h3.x = 180;
    h3.y = 150;
    h3.s = 0.6;

    h4 = new House();
    h4.x = 240;
    h4.y = 150;
    h4.s = 0.4;

    h5 = new House();
    h5.x = 280;
    h5.y = 150;
    h5.s = 0.2;

    drawHouse(h1);
    drawHouse(h2);
    drawHouse(h3);
    drawHouse(h4);
    drawHouse(h5);
}

void drawHouse(House h) {
    rect(h.x, h.y-100*h.s,100*h.s,100* h.s);
    triangle(h.x, h.y-100*h.s,(h.x+50*h.s), h.y-150*h.s,
             (h.x+100*h.s), h.y-100*h.s);
    rect((h.x+35*h.s), h.y-40*h.s,30* h.s,40*h.s);
    point((h.x+55*h.s), h.y-20*h.s);
}
```

Notice how the houses are each stored in variables **h1** through **h5**. That is, the **x**, **y** and **s** data is all kept together for each individual house and stored in their own unique House variable.

Notice as well how the **House** objects are passed as parameters into the **drawHouse** procedure. Within the procedure, each house is referred to as **h**, as indicated by the parameter name. Then, inside the procedure, we simply access the **x**, **y** or **s** component of the house by using **h.x**, **h.y** or **h.s**, respectively.

Example:

Here is how we could use a **Car** data structure in a modification of our accelerating car example that uses two cars. Notice how the use of the data structure simplifies the code when multiple cars are used.

```
Car    myCar, yourCar;      // The two cars to move around

class Car {                 // Definition of what a "Car" actually is
  int      x, y;
  float    speed;
  int      direction;
}

void setup() {              // Initialize the cars by setting the values of their components
  size(600,300);
  myCar = new Car();
  myCar.x = 0;
  myCar.y = 300;
  myCar.speed = 0;
  myCar.direction = 1;

  yourCar = new Car();
  yourCar.x = 300;
  yourCar.y = 300;
  yourCar.speed = 0;
  yourCar.direction = -1;
}

void draw() {               // Repeatedly draw and move each car
  background(255,255,255);
  drawCar(myCar);
  drawCar(yourCar);

  moveCar(myCar);
  moveCar(yourCar);
}


// Draw the car that is passed in as a parameter
void drawCar(Car aCar) {
  fill(150,150,150);
  rect(aCar.x, aCar.y-30, 100, 20);
  quad(aCar.x+20,aCar.y-30, aCar.x+30,aCar.y-45, aCar.x+55,aCar.y-45, aCar.x+70,aCar.y-30);

  fill(0,0,0);  // black
  ellipse(aCar.x+20, aCar.y-10, 20, 20);
  ellipse(aCar.x+75, aCar.y-10, 20, 20);
  fill(255,255,255); // white
  ellipse(aCar.x+20, aCar.y-10, 10, 10);
  ellipse(aCar.x+75, aCar.y-10, 10, 10);
}

// Move the car that is passed in as a parameter
void moveCar(Car aCar) {
  aCar.x = int(aCar.x + aCar.speed*aCar.direction);

  if (mousePressed)
    aCar.speed = min(100, aCar.speed + 0.10);
  else
    aCar.speed = max(0, aCar.speed - 0.10);

  if (abs(aCar.x+50 - mouseX) <= aCar.speed)
    aCar.speed = 0;

  if (mouseX < aCar.x+50)
    aCar.direction = -1;
  else
    aCar.direction = 1;
}
```

Creating an instance of a data structure that has many instance variables, may require many lines of code. For example, creating and initializing **myCar**, in our previous example required 5 lines of code as follows:

```
  myCar = new Car();
  myCar.x = 0;
  myCar.y = 300;
  myCar.speed = 0;
  myCar.direction = 1;
```

In Processing (and Java) we can significantly reduce the amount of code that we need to write when we create and initialize objects by making use of something called a ***constructor***.

> *A* ***constructor*** *is a special procedure that is automatically called to initialize a new object.*

In fact, the parentheses that appear when we do **new Car()** indicate that **Car()** is in fact a kind of procedure or function. This is actually a special kind of function known as the ***default constructor***. What it actually does is that it creates and returns a new fully-initialized object. That is, it reserves space for the data structure's components and sets them all to a value of "zero".

In our above example, however, we wanted to set the **y** value of the car to **300** and the **direction** to **1** … we did not want zeros. In the **yourCar** variable, we further set the **x** value to **300** and **direction** to **-1**. The point is … each object that we create will often have their own unique initial values. In a way, the idea is analogous to building our own computer … we would like to have control to configure our own system with our choice of internal components (i.e., we pick the hard drive, the video card, the motherboard, the monitor, etc..).

We are allowed to write our own constructor procedures so that we can supply our own initial values for our objects. Making a constructor is almost identical to defining a function that takes a bunch of parameters (i.e., one for each of the data structure's instance variables) and then uses these parameters to set the instance variable. The constructor must be written within the data structure's class definition as follows:

```
class Car {                        // Definition of what a "Car" actually is
  int      x, y;
  float    speed;
  int      direction;

  Car(int p1, int p2, float p3, int p4) {   // a constructor
    x = p1;
    y = p2;
    speed = p3;
    direction = p4;
  }
}
```

Notice that the constructor's name is identical to the class name (always uppercase letter to start). Also, notice how there is one parameter for each of the 4 instance variables. The types of the parameters must match the types of the instance variables. The names of the parameters (i.e., **p1**, **p2**, **p3** and **p4**) are arbitrary, but they must be unique from one another and MUST NOT be the same as any instance variable names.

The code for the body of the constructor is simple. It simply sets each instance variable to have the value of its corresponding parameter.

So what does this all mean ? It means that once we define this constructor, we can then call the constructor with the parameter values that we want to have set in the instance variables. Below shows the previous **setup()** code before we had the constructor … along with the new code that makes use of the constructor:

Without the constructor:

```
Car    myCar, yourCar;

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

void setup() {
  size(600,300);
  myCar = new Car();
  myCar.x = 0;
  myCar.y = 300;
  myCar.speed = 0;
  myCar.direction = 1;

  yourCar = new Car();
  yourCar.x = 300;
  yourCar.y = 300;
  yourCar.speed = 0;
  yourCar.direction = -1;
}
```

With the constructor:

```
Car    myCar, yourCar;

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

  Car(int p1, int p2, float p3, int p4) {
    x = p1;
    y = p2;
    speed = p3;
    direction = p4;
  }
}

void setup() {
  size(600,300);
  myCar = new Car(0, 300, 0, 1);
  yourCar = new Car(300, 300, 0, -1);
}
```

Notice the drastic reduction of code within the **setup()** procedure. The saving in code space can be much more drastic when multiple kinds of objects are used together. For example, consider that we created constructors for the **FullName**, **Address** and **BankAccount** objects:

```
class FullName {
   String     firstName, middleName, lastName;

   FullName(String p1, String p2, String p3) {
     firstName = p1;
     middleName = p2;
     lastName = p3;
   }
}
```

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

  Address(FullName p1, int p2, String p3, String p4, String p5, String p6) {
    name = p1;
    streetNumber = p2;
    streetName = p3;
    city = p4;
    province = p5;
    postalCode = p6;
  }
}
```

```
class BankAccount {
   Address    owner;
   int        accountNumber;
   float      balance;

  BankAccount (Address p1, int p2, float p3) {
    owner = p1;
    accountNumber = p2;
    balance = p3;
  }
}
```

Once we make such definitions, notice the significant simplification in code:

Without the constructor:

```
BankAccount  b;
Address      a;
FullName     n;

n = new FullName();
n.firstName = "Patty"
n.middleName = "Ohh"
n.lastName = "Lantern"

a = new Address();
a.name = n;
a.streetNumber = 187
a.streetName = "Oak St."
a.city = "Ottawa"
a.province = "ON"
a.postalCode = "K6S8P2"

b = new BankAccount();
b.owner = a;
b.accountNumber = 829302
b.postalCode = 2319.67
```

With the constructor:

```
BankAccount  b;
Address      a;
FullName     n;

n = new FullName("Patty", "Ohh", "Lantern");
a = new Address(n, 187,"Oak St." ,"Ottawa","ON","K6S8P2");
b = new BankAccount(a, 829302, 2319.67);
```

So, constructors can be a significant factor in keeping your code simple. We will discuss constructors in more detail in COMP1406.

Example:

Recall our example that allowed us to throw a ball around in the window:

```
int          x, y;        // location of the ball at any time
float        direction;   // direction of the ball at any time
boolean      grabbed;     // true if the ball is being held
float        speed;       // the ball’s speed

final int    RADIUS = 40;              // the ball’s radius
final float  ACCELERATION = 0.10;      // acceleration/deceleration amount

void setup() {
  size(600,600);
  x = width/2;
  y = height/2;
  direction = random(TWO_PI);
  grabbed = false;
  speed = 10;
}

void draw() {
  background(0,0,0);
  ellipse(x, y, 2*RADIUS,2*RADIUS);

  // move the ball forward if not being held
  if (!grabbed) {
    x = x + int(speed*cos(direction));
    y = y + int(speed*sin(direction));
  }
  else {
    x = mouseX;
    y = mouseY;
  }

  speed = max(0, speed - ACCELERATION);

  if ((x+RADIUS >= width) || (x-RADIUS <= 0))
    direction = PI - direction;
  if ((y+RADIUS >= height) || (y-RADIUS <= 0))
    direction = -direction;
}

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;
}
```

How could we adjust the code to use a **Ball** data structure ? What information in this program, will make up the attributes of the ball ?

It is easy to see that 4 of the 5 variables actually represent the ball's state … that is, its (x, y) position, direction, and speed. Whether the ball has been grabbed or not is not necessarily part of the Ball's attributes. That is, a ball does not need to know whether or not it has been grabbed. For example, we may have an application in which the balls are moving around without the ability to be grabbed. So, here is what our **Ball** data structure would look like along with its constructor:

```
class Ball {
    int         x, y;       // location of the ball at any time
    float       direction;  // direction of the ball at any time
    float       speed;      // the ball's speed

    Ball(int p1, int p2, float p3, float p4) {
          x = p1;
          y = p2;
          direction = p3;
          speed = p4;
    }
}
```

Assuming that we define this class in our program, the remainder of the program can now be adjusted as follows. Notice the difference between the code before the Ball class is used and after:

Without Ball Class Definition	**With** Ball Class Definition

`final int   RADIUS = 40;`
`final float ACCELERATION = 0.10;`
	`class Ball {      // as defined earlier`
	`  ...`
	`}`
	`final int   RADIUS = 40;`
	`final float ACCELERATION = 0.10;`

`int         x, y;`
`float       direction;`
`float       speed;`
	`Ball        b;     // the Ball`

`boolean     grabbed;`
 	
`void setup() {`
`  size(600,600);`
	`boolean     grabbed;`
	
	`void setup() {`
	`  size(600,600);`

`  x = width/2;`
`  y = height/2;`
`  direction = random(TWO_PI);`
`  speed = 10;`
	`  b = new Ball(width/2, height/2,`
	`                random(TWO_PI), 10);`

`  grabbed = false;`
`}`
 	
`void draw() {`
`  background(0,0,0);`
`  ellipse(x, y, 2*RADIUS,2*RADIUS);`
	`  grabbed = false;`
	`}`
	
	`void draw() {`
	`  background(0,0,0);`
	`  ellipse(b.x, b.y, 2*RADIUS,2*RADIUS);`

```
  if (!grabbed) {
    x = x + int(speed*cos(direction));
    y = y + int(speed*sin(direction));
  }
  else {
    x = mouseX;
    y = mouseY;
  }

  speed = max(0, speed - ACCELERATION);

  if ((x+RADIUS>=width)||(x-RADIUS<=0))
    direction = PI - direction;
  if ((y+RADIUS>=height)||(y-RADIUS<=0))
    direction = -direction;
}

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;
}
```

```
  if (!grabbed) {
    b.x=b.x+int(b.speed*cos(b.direction));
    b.y=b.y+int(b.speed*sin(b.direction));
  }
  else {
    b.x = mouseX;
    b.y = mouseY;
  }

  b.speed = max(0, b.speed-ACCELERATION);

  if ((b.x+RADIUS>=width)||(b.x-RADIUS<=0))
    b.direction = PI - b.direction;
  if ((b.y+RADIUS>=height)||(b.y-RADIUS<=0))
    b.direction = -b.direction;
}

void mousePressed() {
  if (dist(b.x, b.y,mouseX,mouseY)<RADIUS)
    grabbed = true;
}

void mouseReleased() {
  if (grabbed) {
    b.direction = atan2(mouseY - pmouseY,
                        mouseX - pmouseX);
    b.speed = int(dist(mouseX, mouseY,
                       pmouseX, pmouseY));
  }
  grabbed = false;
}
```

Notice how the ball's attributes are all now defined inside the **Ball** object so that less variables are needed in the main program. Also, the remainder of the code simply requires **b.** in front of these attributes in order to go into the object to get their values.

While it seems as though we are writing a little bit more code now, the advantage of creating this **Ball** data structure will be more clear later.

Chapter 5

Lists, Arrays and Searching

What is in This Chapter ?

When solving problems, we often deal with data that has been collected together. We often must sift through collections of information to find answers. This chapter discusses how data can be collected together into ***Arrays*** and also the various ways that we can ***search*** through the data efficiently to find what we want.

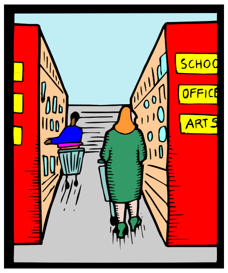
5.1 Collecting Data Using Lists

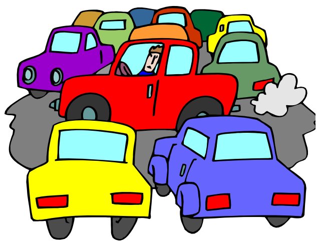
As we have seen in the last section, it is easy to define our own data structures. In fact, it is such common practice to define data structures that there is an entire 2nd year course on Data Structures and Algorithms. We will not discuss the deeper concepts related to data structures in this course.

Obviously, we can define a particular data structure to suit our own needs. There are, however, some common well-known kinds of data structures that are fundamental to the development of more useful algorithms. Such data structures have "similar patterns" of what they store and how they are used. When we have two or more particular data structures that fall under the same "pattern" of usage, we group all such data structures into a category and denote such a category as an ***abstract data type*** (ADT).

One such abstract data type is called a ***Collection*** which represents a group of objects that are treated as a single entity. Collections appear in real-life 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





A more specific kind of collection is called an ***Ordered Collection*** or ***List***. The concept of a list is intuitive to most of us. We know it as a bunch of items that are placed in some kind of order (e.g., ordered in a first-come-first-served basis, or perhaps sorted alphabetically, etc..).

We may think of a list written on a piece of paper, perhaps of items that need to be picked up at the grocery store. Even though the order of the items is unimportant, it nevertheless has an order (perhaps arbitrary) to the items. Sometimes, a list may also be sorted, perhaps alphabetically or by price or date. Such a kind of ordering would make the list a ***Sorted List***.

For the remainder of this discussion, we will assume that the word **List** represents an unordered list of items, unless otherwise stated.

When using a list, there are certain standard functions and procedures that are expected to be available. That is, we should be able to create a new list, add something to the list, remove something from the list etc… One of the most common operations to perform on a list is to iterate (or traverse) through its elements one-by-one in order. Let us look at some examples that build up some lists of items and then do something simple with the list data.

Example:

Consider working as an assistant at a library. Imagine that someone was handing books to you and you needed to place them side-by-side on an empty shelf. What represents the *collection* data structure in this situation ?

The shelf is the collection, and in fact … it is an ordered list since the books will eventually have an order once they are on the shelf (i.e., in the list). Assuming that we had to find the empty shelf on our own, how would we write an algorithm for stocking the shelf, assuming that there is a shelf with enough space for all the books ?

```
Algorithm: PlaceBooks

1.    shelf ← new List
2.    while there are more books {
3.        book ← get the next book
4.        add book to shelf
      }
```

The algorithm is simple and logical, but it does not indicate where to add the books on the shelf. It may not matter. In this case, step 4 may simply require adding the book to the end of the shelf, beside the last one added.

However, if the books are to be sorted by their call numbers, then the order will matter. It will require us to insert the book at a specific position on the shelf. We can be a little more specific in our code by making use of the call number:

```
Algorithm: PlaceSortedBooks

1.    shelf ← new List
2.    while there are more books {
3.        book ← get the next book
4.        callNumber ← look at the book's call number
5.        insert book into shelf at the appropriate callNumber location
      }
```

Of course, step 5 can always be clarified further, but the algorithm as shown is understandable at a high level.

Example:

Now assume that we want to transfer all the books from an existing shelf to an empty shelf of the same size. How would we adjust the algorithm to do this, assuming that we are given two parameters to the algorithm representing the shelves ?

```
Algorithm: TransferBooks
    shelf1:     shelf containing some books to be transferred
    shelf2:     an empty shelf to place the books onto

1.      for each book on shelf1 {
2.          remove book from shelf1
3.          add book to shelf2
        }
```

The algorithm is straight-forward, and again … a similar version for sorted books can be written.

Example:

The above example assumes that there is enough space on **shelf2** to hold the books that were on **shelf1**. How could we adjust the algorithm to account for the situation where **shelf2** is smaller than **shelf1** ?

```
Algorithm: TransferLimitedBooks
    shelf1:     shelf containing some books to be transferred
    shelf2:     an empty shelf to place the books onto

1.      for each book on shelf1 {
2.          if (there is space on shelf2) then {
3.              remove book from shelf1
4.              add book to shelf2
            }
        }
```

As we will see later, there are more details to work out when we begin to program, but for now, it is important to understand how lists can be used in other important examples.

5.2 List Searching Algorithms

In real life, we are often faced with the problem of sifting through information (i.e., data) to determine whether or not a particular type of value lies within the information. We call this the ***searching*** problem, since we are searching through data for a (possibly partial) solution to our problem at hand.

For example, we may look through a list to:

1. determine the existence of a piece of data (e.g., check if a person is on a list)
2. verify that all items satisfy a condition (e.g., is anyone missing from work today)
3. pick an appropriate candidate for our problem (e.g., find an available seat on the bus)
4. determine the most efficient solution (e.g., detect the shortest line at Walmart)
   etc..

A ***linear search*** (or ***sequential search***) is a method of checking every one of the elements in a list, one at a time and in sequence, until a value is found that satisfies some particular condition (e.g., until a *match* is found).

When developing algorithms, the simplest approach is often called a "**brute force**" approach, implying that there is a lack of human sensibility in the solution. A "brute force" algorithm is one that is often easy to come up with, but that does not usually consider efficiency or any form of ingenuity.

A linear search is the simplest kind of search since it involves naively looking through the elements in turn for the one that matches the criteria. In all the examples below, we will assume that the items that we are searching through are stored as an array (explained later).

Example:

Consider determining whether or not a person is on a list, perhaps to allow them admittance into a special event. How would you develop an algorithm to do this using a linear search ?

Assume for example that we wanted to find "Patty O. Lantern" on the list shown here. It may be obvious that all we need to do is to start at the top of the list and then work our way down towards the bottom, comparing names along the way.

But how do we explain this procedure with an algorithm ?
Here is one way …

```
Algorithm: IsOnList
        names:              the list (e.g., array) to search through
        searchName:         the name to search for

1.      found ← false
2.      for each name in the names list {
3.          if (name = searchName) then
4.              found ← true
        }
5.      print found
```

Name
Bob E. Pins
Sam Pull
Mary Me
Jim Class
Patty O. Lantern
Robin Banks
Shelly Fish
Barb Wire
Tim Burr
Dwayne D. Pool
Hugh Jarms
Ilene Dover
Frank N. Stein
Ruth Less

Notice that the **for** loop checks all the people on the list one-by-one. Notice the use of the Boolean variable **found**. This is called a boolean ***flag*** in computer science. It is analogous to the little flag on mailboxes that the mailperson lifts up to let people know when their mail has arrived.

Notice how the *flag* is down (i.e., **false**) before the loop starts and that it only gets raised (i.e., set to **true**) when the **name** matches the **searchName**.

The above algorithm always takes ***linear time*** … that is, in the worst case it may require us to check all of the names on the list (i.e., the last name on the list matches).

In practice however, it is possible to exit the loop when a match is found. After all, why keep looking through the rest of the names when we have already found it. Consider using a **while** loop where the *flag* is the loop condition:

```
Algorithm: IsOnList2
        names:              the list (e.g., array) to search through
        searchName:         the name to search for

1.      found ← false
2.      while ((found = false) AND (there are more names to check)) {
3.          name ← get the next name in the names list
4.          if (name = searchName) then
5.              found ← true
        }
6.      print found
```

Notice now how the **false** value at the beginning ensures that we enter into the **while** loop. Then, once the name is found, the *flag* is set and the loop will stop right away. This is more efficient in practice, although in the worst case, it may still require us to check all of the people on the list. Notice as well that the loop ends when no more names are available.

Example:

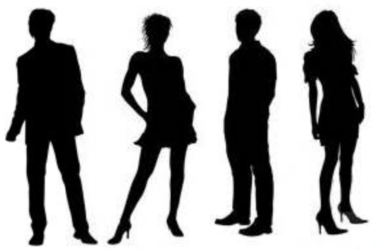
Consider now determining whether or not all items in a list satisfy some condition. For example, how would we write an algorithm that determines whether or not **everyone** on a list is 18 years of age or older ? This is a kind of verification algorithm that should return **false** if just one "under age" person is found.

Name	**Age**
Bob E. Pins	25 ✓
Sam Pull	24 ✓
Mary Me	31 ✓
Jim Class	54 ✓
Patty O. Lantern	62 ✓
Robin Banks	18 ✓
Shelly Fish	17 ×
Barb Wire	21
Tim Burr	26
Dwayne D. Pool	47
Hugh Jarms	36
Ilene Dover	42
Frank N. Stein	13
Ruth Less	71

Likely we'll begin as before checking each person's age one-by-one. But what are we looking for ? We are looking for someone who is under the age of 18.

Do we need to check every person, or can we jump out of the loop at any time ? Here is the algorithm:

```
Algorithm: AllAdults
        people:         the list (e.g., array) to search through

1.      allAdults ← true
2.      for each person in the people list {
3.              if (person.age < 18) then
5.                      allAdults ← false
        }
6.      print allAdults
```



The structure of this search algorithm is quite similar to our previous example. Notice however, that we began with an assumption that all in the list are adults and only set the *flag* to **false** when there is someone found who is under age. Do you understand why we had to switch the *flag* around like this ?

Example:

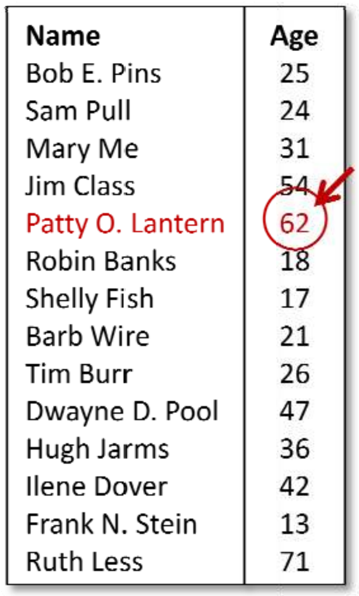
Now let us go back to our algorithm that searches for a name on a list. Perhaps to get into the special event, we need to show our ID at the door so that our age can be verified.

How would we adjust the algorithm so that it didn't simply return **true** or **false** whether or not a person is on the list but in fact returned something about the person, such as their **age** ?

What is different ? Do we still need a boolean *flag* ?

Here is a solution:

```
Algorithm: FindAgeOfPerson
        people:            the list (e.g., array) to search through
        searchName:        the name to search for

1.      for each person in the people list {
2.            if (person.name = searchName) then
3.                  age ← person.age
        }
4.      print age
```



Name	Age
Bob E. Pins	25
Sam Pull	24
Mary Me	31
Jim Class	54
Patty O. Lantern	62
Robin Banks	18
Shelly Fish	17
Barb Wire	21
Tim Burr	26
Dwayne D. Pool	47
Hugh Jarms	36
Ilene Dover	42
Frank N. Stein	13
Ruth Less	71

Notice now that we did not use a boolean *flag* but instead we stored the **age** of the person that we found in the list so that we could print it later. Of course, if all we wanted to do was print the **age**, we could have done it within the loop:

```
Algorithm: FindAgeOfPerson2
        people:            the list (e.g., array) to search through
        searchName:        the name to search for

1.      for each person in the people list {
2.            if (person.name = searchName) then
3.                  print (person.age)
        }
```

We can also even adjust the code to use a **while** loop so as to allow the loop to exit early:

```
Algorithm: FindAgeOfPerson3
        people:            the list (e.g., array) to search through
        searchName:        the name to search for

1.      found ← false
2.      while ((found = false) AND (there are more people to check))  {
3.            person ← get the next person in the people list
4.            if (person.name = searchName) then {
5.                  print (person.age)
6.                  found ← true
              }
        }
```

Notice now that the algorithm is efficient in that it prints out the **age** and quits the loop when the person has been found.

One problem, however, with our above solutions is that they do not handle the situation in which the person is not found in the list. In the 2nd and 3rd case, this may not be a problem since nothing is printed. In the 1st case however, the **age** is printed on the last line. But this **age** is only set when the person is found. What is the **age** set to when the person is not found ? This depends on the computer language being used. Often, the **age** will default to zero, and so **0** will be printed. It might be a good idea to specify the "default" age at the top of the algorithm … perhaps **age = -1** or something like that to indicate that it is invalid.

Example:

Another very common task when searching through a set of values is to find the maximum or minimum. Can you write an algorithm to find the age of the oldest person out of a set of people ?

What do you need to keep track of as you are figuring out who has the maximum age ?

```
Algorithm: FindOldest
        people:                 the list (e.g., array) to search through

1.      oldestSoFar ← 0
2.      for each person in the people list {
3.              if (person.age > oldestSoFar) then
4.                      oldestSoFar ← person.age
        }
5.      print oldestSoFar
```

Notice how the **oldestSoFar** variable is used as a "record keeper" to remember the **age** of the person who is currently the oldest as we search systematically through the list. We only change that value once we find someone older.

To find the youngest person, the algorithm is nearly identical. Do you know what changes ?

```
Algorithm: FindYoungest
        people:         the list (e.g., array) to search through

1.      youngestSoFar ← 200
2.      for each person in the people list {
3.              if (person.age < youngestSoFar) then
4.                      youngestSoFar ← person.age
        }
5.      print youngestSoFar
```

As you can see, the code is basically the same, except for the minor change in the condition as well as the initial starting value for the **youngestSoFar**. Why did we set this to such a large number like **200** ? What would have happened if we left it at **0** ? Do you understand ?

It is important to choose a number that we know to be larger than any other possible age of the people that we are searching through. In our times, 200 is a reasonable number since all people will have an age lower than this. However, just a few thousand years ago, people lived longer than 200 years old. For example, if you recall the historical account of Noah (the man that built the ark with all the animals on it) … history records that he lived 950 years old! People actually lived a long time back then .. the oldest recorded was 969 (Noah's grandfather whose name was Methuselah).

How would our algorithm perform if used on people who lived that long ? Well, likely all people will be older than 200 years and therefore the algorithm would never set the **youngestSoFar** variable and so the answer would always be 200 … which is wrong.

So, it is always "better to be safe than sorry" by choosing a very large value that we know to be above all possible values in our problem. Recorded history tells us therefore that any number 1000 or more would be sufficient to ensure that at least one of our people are below that age. Some computer languages have predefined constants representing the maximum integer or minimum integer. Processing and JAVA, for example, have constants that represent the largest and smaller integers that can be stored: **Integer.MAX_VALUE** and **Integer.MIN_VALUE**.

These would be good value to use if we are trying to find maximum and minimum numerical values. Whenever we find ourselves in need of determining the largest/maximum or smallest/minimum value of a list, the templates above will always work.

Sometimes we need to keep track of additional information as we find the maximum and minimum. For example, what if we wanted to know the **name** of the oldest person, not just the age ? What do you need to keep track of as you are figuring out who has the maximum age ?

```
Algorithm: FindNameOfOldest
        people:         the list (e.g., array) to search through

1.      oldestAge ← 0
2.      oldestName ← "unknown"
3.      for each person in the people list {
4.          if (person.age > oldestAge) then {
5.              oldestAge ← person.age
6.              oldestName ← person.name
            }
        }
7.      print oldestName
```

Notice how both the **age** and the **name** of oldest person is maintained throughout the loop. The **oldestAge** is used to compare against each person's age while the **oldestName** is simply stored as the name of the person who had the **oldestAge** so far.

As before, we would have to choose some kind of default **oldestName** for the situation in which no oldest person is found. However, this would only occur when all people are 0 years of age or more likely when there is nobody on the list. Provided that we choose a proper starting **oldestAge**, then the default **oldestName** will only be an issue in the case where the list is empty.

We could actually simplify the code to make use of the entire **person** data structure (with the **name**, **age**, etc..) as opposed to just the **name** as follows:

```
Algorithm: FindNameOfOldest2
        people:         the list (e.g., array) to search through

1.      oldest ← the first person in the people list
2.      for each person in the people list {
3.          if (person.age > oldest.age) then
4.              oldest ← person
        }
5.      print oldest.name
```

Notice now that only one variable is needed because it is an entire data structure that contains the **name** & **age** of the person who is the **oldest**. (Remember, a data structure is really just a grouping together of data). In fact, if we wanted, on line 5 we could access or print out any (or all) of that **oldest** person's information because it is readily available within the data structure.

Notice that line 1 set the **oldest** to initially be the first person in the list. Beginning with the first value in a list is a typical when finding a maximum or minimum value as opposed to beginning with a value of **null**. That is because the very first person in the list would be compared in line **3** with **oldest.age**. If oldest is **null**, then **oldest.age** would be undefined and the code would not make sense.

Example:

In our examples above, we did not consider the situation in which people have the same name and/or the same age. For example, consider what happens when we are searching for a name in a list that has duplicates.

In the case where we are trying to find out whether a particular person is on the list, the algorithm need not worry. Also, if we are trying to determine whether or not everyone on the list is an adult, duplicates do not affect the algorithm either.

Consider people playing a game one or more times and recording their score on a list. In this case, a person may appear more than once and their associated score is likely unique each time.

What would be the result of the following algorithm if the **searchName** is "Patty O. Lantern":

```
Algorithm: FindScoreOfPerson
        people:        the list (e.g., array) to search through
        searchName:    the name to search for

1.      for each person in the people list {
2.           if (person.name = searchName) then
3.                score ← person.score
        }
4.      print score
```

Name	Score
Hugh Jarms	36
Bob E. Pins	25
Shelly Fish	17
Sam Pull	24
Mary Me	31
Jim Class	54
Patty O. Lantern	62
Robin Banks	18
Shelly Fish	17
Barb Wire	21
Patty O. Lantern	31
Tim Burr	26
Dwayne D. Pool	47
Hugh Jarms	31
Ilene Dover	42
Frank N. Stein	13
Patty O. Lantern	16
Ruth Less	71

Which score gets printed ? The last one does. Careful examination of the code shows that the **score** variable is set three times for the list shown above, the 1st and 2nd values are overwritten by the 3rd value of **16**.

We can actually adjust the algorithm to list all the scores for that person by moving the print procedure from step 5 to step 4 like this: **print (**score of **person)**.

Example:

How could we alter the algorithm so that it displayed the average score for the person ?

```
Algorithm: AverageScoreOfPerson
        people:        the list (e.g., array) to search through
        searchName:    the name to search for

1.      score ← 0
2.      count ← 0
3.      for each person in the people list {
4.           if (person.name = searchName) then {
5.                score ← score + person.score
6.                count ← count + 1
             }
        }
7.      print score / count
```

Name	Score
Hugh Jarms	36
Bob E. Pins	25
Shelly Fish	17
Sam Pull	24
Mary Me	31
Jim Class	54
Patty O. Lantern	62
Robin Banks	18
Shelly Fish	17
Barb Wire	21
Patty O. Lantern	31
Tim Burr	26
Dwayne D. Pool	47
Hugh Jarms	31
Ilene Dover	42
Frank N. Stein	13
Patty O. Lantern	16
Ruth Less	71

+

Notice how the **score** is totaled each time we notice the person in the list. Also, do you understand why we needed to also maintain a **count** of how many times the person's name appeared on the list ? It is necessary in order to compute the average.

Example:

How about finding the highest score where there are multiple people with the same score ?

If you remember, the first step to problem solving is to *understand* the problem. So what should the answer be ? Well, three people in our example here have the highest score. What should the output be ? We have some choices:

1. Print the first person who received the highest score.
2. Print the last person who tied the highest score.
3. Print all three who have the highest score.

What would the algorithm look like in each case ? For the first case, the format is identical to the one that found the oldest person:

```
Algorithm: HighestScore
        people:          the list (e.g., array) to search through

1.      highestScore ← 0
2.      highestName ← "unknown"
3.      for each person in the people list {
4.              if (person.score > highestScore) then {
5.                      highestScore ← person.score
6.                      highestName ← person.name
                }
        }
7.      print highestName
```

Name	Score
Hugh Jarms	36
Bob E. Pins	25
Shelly Fish	47
Sam Pull	24
Mary Me	91
Jim Class	54
Patty O. Lantern	62
Robin Banks	18
Shelly Fish	17
Barb Wire	21
Patty O. Lantern	91
Tim Burr	26
Dwayne D. Pool	47
Hugh Jarms	86
Ilene Dover	91
Frank N. Stein	13
Patty O. Lantern	16
Ruth Less	71

For the 2nd case, we just need to change the condition in step 5 to (**score** >= **highestScore**). The 3rd case is a little trickier. Can you explain why the following code will NOT work ?

```
highestScore ← 0
highestName ← "unknown"
for each person in the people list {
        if (person.score >= highestScore) then {
                highestScore ← person.score
                print (person.name)
        }
}
```

This algorithm would print out every person whose score exceeded the current high score as we went through the list. Here would be the names printed:

```
Hugh Jarms, Shelly Fish, Mary Me, Patty O. Lantern, Ilene Dover
```

Now, while the last three names are correct, the first two do not share the same high score. What is wrong with the algorithm ? We cannot simply print out the person whenever they "beat" the currently best high score. We need to know that the currently high score is the actual highest score before we print. So, we will need to print at the end, but somehow remember all those who had the highest score.

To do this, we will need to keep a new list of people who share the high score. When do we add to the list ? Do we ever need to reset the list ? Think about it.

Here is a revised (and correct) solution:

```
Algorithm: HighestScore
        people:                 the list (e.g., array) to search through

1.      highestScore ← 0
2.      highestList ← new List
3.      for each person in the people list {
4.              if (person.score > highestScore) then {
5.                      highestScore ← person.score
6.                      highestList ← new List
                }
7.              if (person.score = highestScore) then
8.                      add person.name to highestList
        }
9.      print highestList
```

Notice that whenever we find a new highest score, the **highestList** of people who had the previous highest score is emptied out (step 7). Then in step 9, this person is added to that newly emptied **highestList**. Whenever someone else is found who shares that highest score, he/she is simply added to the **highestList**.

Example:

Here is a tougher one now. What if we were on our way to an autoshow but before we left our roommate asked us (for some strange/unknown reason) to determine the most popular color of car at the autoshow.

Now how do we approach the problem ?

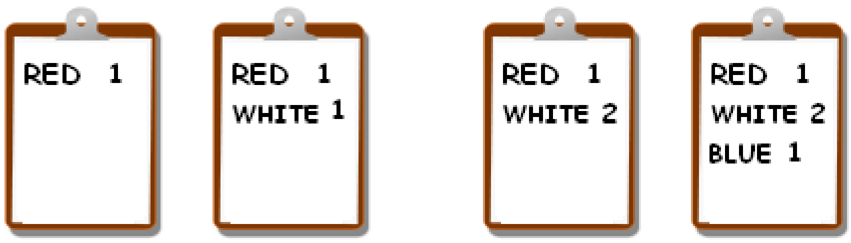
Well think of real life. Assuming that there were hundreds of cars and that your memory is not perfect, you would likely bring with you a piece of paper (perhaps on a clipboard) so that you can keep track of the colors of cars that you find.

When you enter the autoshow and see the first car, you would look at its **color** and then likely write down the color on the paper and maybe put the number 1 beside it.

Assume that you went to the next car and that it was a different color. You would write that new color down too along with a count of 1. If you find a car with the same color again, you would just increment its count.

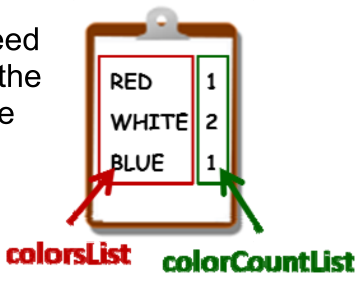
Below is a picture showing how your clipboard gets updated as you encounter car colors in the order of red, white, white, blue, etc..:

etc..



etc..

If we were to take this approach in our programming solution, then we need to realize that each color that we write down on our clipboard will have a single number associated with it at all times (representing the **count** of the number of times that color appeared).



The clipboard/paper itself represents our list of colors so we would need a list to store that. In fact, since we need a list of **counts** along with the list of **colors**, we will need ***two*** lists… one to store the **colors** and one to store the **counts**.

At this point, we can do a partial algorithm that will count up the number of times that each color appears at the show as follows:

```
Algorithm: CarColorCount
        cars:                       the list (e.g., array) to search through

1.      colorsList ← new List
2.      colorCountList ← new List
3.      for each car in the cars list {
4.            if (car.color is not in the colorsList) then
5.                  add car.color to colorsList
6.            add 1 to colorCountList for the color corresponding to car.color
        }
```

Once this part of the algorithm has been completed, we have two lists filled in showing the colors at the autoshow and their corresponding counts.

To find the most popular color, we simply need to find out which of these colors has the largest value (i.e., the maximum count). We will assume that there are no duplicates (or that only the first one with the maximum count is the answer).

The solution would then be to first count the colors using the above algorithm and then apply a max/min algorithm similar to the one that found the name of the oldest person as we did before:

```
Algorithm: MostPopularColor
        cars:                       the list (e.g., array) to search through

1.      perform algorithm CarColorCount()
2.      bestCountSoFar ← 0
3.      bestColorSoFar ← unknown
4.      for each color in the colorsList {
5.            count ← count in colorCountList corresponding to color
6.            if (count > bestCountSoFar) then {
7.                  bestCountSoFar ← count
8.                  bestColorSoFar ← color
              }
        }
9.      print bestColorSoFar
```

Notice that the structure is similar to the standard min/max template except that we performed a kind of pre-processing stage in step 1 to count the colors.

Alternatively, we could have combined the color counting and max-finding sub-algorithms together:

```
Algorithm: CombinedMostPopularColor
        cars:                    the list (e.g., array) to search through

1.      colorsList ← new List
2.      colorCountList ← new List
3.      bestCountSoFar ← 0
4.      bestColorSoFar ← unknown

5.      for each car in the cars list {
6.              if (car.color is not in the colorsList) then
7.                      add car.color to the colorsList
8.              add 1 to colorCountList for the color corresponding to car.color
9.              count ← count in colorCountList corresponding to car.color

10.             if (count > bestCountSoFar) then {
11.                     bestCountSoFar ← count
12.                     bestColorSoFar ← car.color
                }
        }
13.     print bestColorSoFar
```

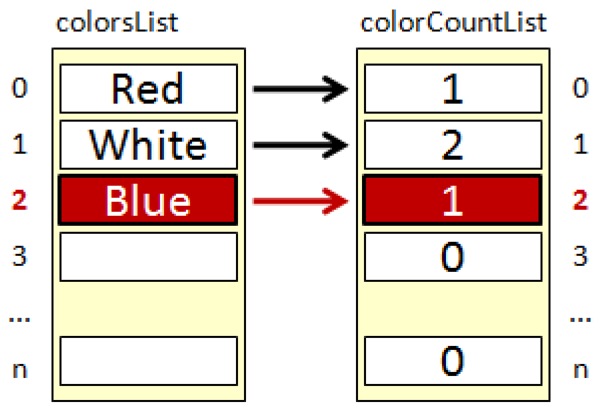
5.3 Arrays

In most of our list examples, we were not overly concerned about particular positions of items in the list. That is, we usually iterated through the items in the list one at a time, but were not concerned about the number of the item within the list. When we used a **while** loop, we simply asked to "get the next item in the list". Therefore, we were essentially treating the list as if it was unordered (i.e., as if the order did not matter):

```
for each person in the people list { … }
for each car in the cars list { … }
name ← get the next name in the names list
```

However, in our last example, we wanted to maintain a count for each color in the list. When we encountered a particular color, we needed to find that color in the **colorsList** and add 1 to the corresponding color in the **colorCountList** →

To do this, we need to make sure that we update the correct counter each time. So, we need to know the position of the color in the **colorsList** and update the counter at the ***SAME*** position in the **colorCountList**.

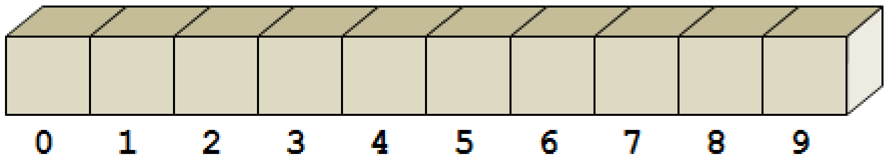


A list in which the order of the items is important to know, is called an ***ordered list***. Some programming languages have pre-defined data types to represent ordered lists with convenient methods such as **add**, **remove**, **insertItemAtPosition**, **getItemAtPosition**, etc... However, not all languages have such a convenient data type readily available.

All the popular programming languages do, however, have a fixed data type called an ***array***.

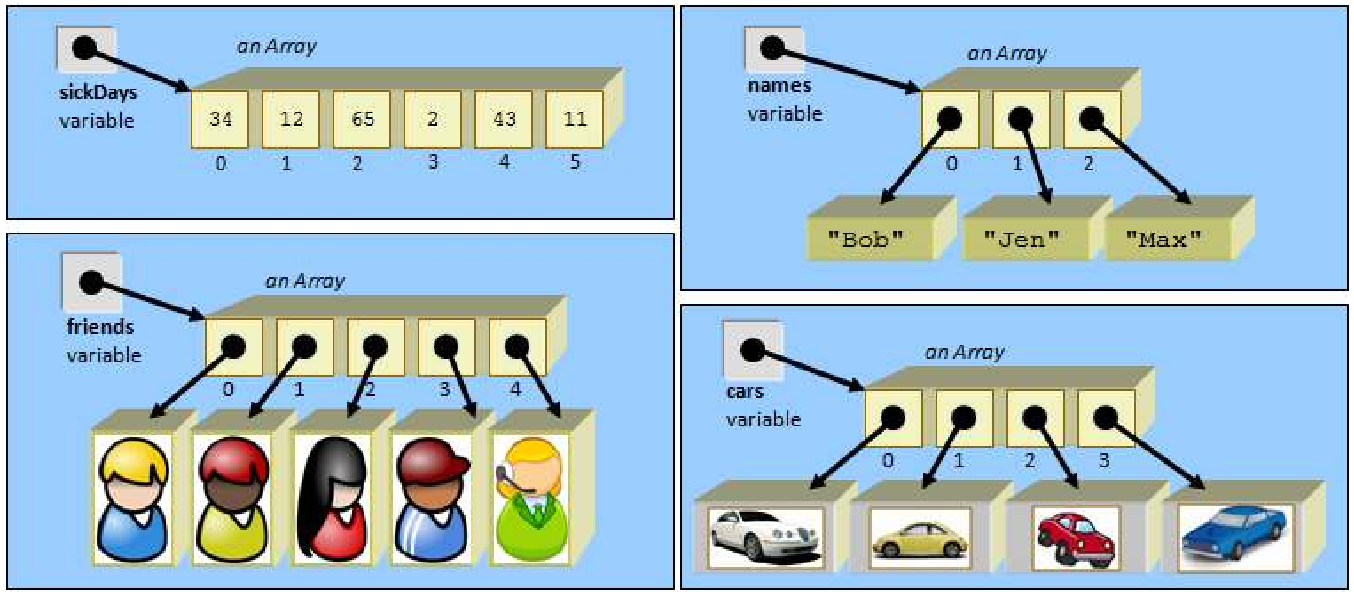
*An **array** is a collection of data items that can be selected by indices (or locations).*

So, arrays are a means of "gluing" a bunch of variables side-by-side in some specified order.

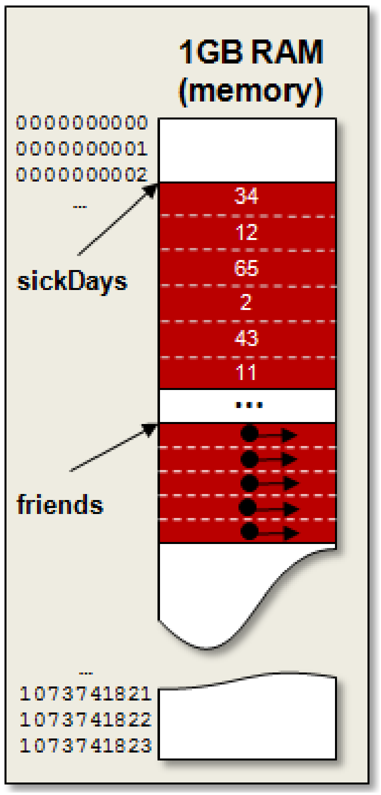


Each item (also known as ***element***) in an array is stored at a location which is specified by an **index**. The index is an integer that identifies the location within the sequence of items. Indices start at 0. Arrays are fixed-sized collections in that they can hold a fixed number of items … that is … they don't grow or shrink … they have a fixed size. In the above image, the array holds exactly 10 items, so the indices go from **0** through **9**. An index of **10** or higher would be out of the array's "bounds", and would therefore be invalid. We often refer to the "size" of the array as its **length**. The size/or length of the array is NOT the number of items that we have put inside it, rather, it is the ***capacity*** of the array (i.e., the maximum number of items that we can put into it). We will use **anArray.length** in our pseudocode to indicate the length of an array which is stored in the **anArray** variable.

Arrays themselves are data types that store a particular kind of item within them. Each item is understood to be of the same type (e.g., all integers, all floats, all Strings, all Cars, etc..). Here are some examples of arrays that hold integers, Strings, Person objects and Car objects:



Notice how each item in the array is of the same type. The array data structure is quite efficient in how it actually stores the items in memory. Once allocated in memory, the array takes up a contiguous (i.e., connected without a break) sequence of memory locations.



If the data being store is simple primitive-like data, then it is stored directly in the array one item after another. The **sickDays** variable above, for example, represents an array with simple integers that would be store sequentially in memory as shown here: →

If, however, the array's data items are a more complex data type, then it is stored differently in a way that depends on the programming language.

For example, in JAVA/Processing, pre-existing objects may be created and then the array would store simply pointers (e.g., memory addresses) to those objects. So an array of items of type **Person**, would store a sequence of pointers to the individual **Person** objects, as shown here in the **friends** array: →

Alternatively, in languages like C for instance, the array would simply reserve contiguous space for the entire series of Person data types by calculating the size of a person data structure and multiplying by the size of the array.

Because array data is stored and accessible in contiguous memory locations, we can actually refer to all of the data by using a single variable name! That is nice. We can use just one variable to refer to all the items, even if there are thousands of them.

Of course, when we want to access a particular item, we need to indicate which one we want. So, we need to supply its index. Also, each time we want to put something into the array we also need to supply the index location.

So, we will use square brackets (i.e., **[ ]**) whenever we refer to a particular item in the array. Thus, we will use the array's variable name, followed by square brackets and place the index number within the brackets as follows:

```
theArray[5]
```

Example:

Consider for example our previous algorithm for determining whether or not someone's name (i.e., **searchName**) is on a list (i.e., **names**). The left side of the table below shows the code as was used with *unordered* lists, while the right side indicates the *array* version.

Unordered List Version	Array Version

found ← false
for each name in the names list {
    if (name = searchName) then
            found ← true
}
print found
	found ← false
	for index i from 0 to names.length-1 {
	    if (names[i] = searchName) then
	            found ← true
	}
	print found

Notice a couple of things with the above code. We are specifying in the **for** loop the index number (stored in variable **i**) of the items in the array. Then, this index is used in the **if** statement condition to access the name at position **i** in the **names** array. You may notice that there is no **name** variable to store the name. Instead, we simply use the entire array and indicate the name by means of the square brackets and index **i**.

So you can see that the code is essentially the same, except that we now need to specify the location of each item in the list by means of its index.

Example:

Recall the example that determined the person with the highest score in the unordered list. Again, below is a comparison of the unordered list version with the array version. In the example, **people** is the parameter containing **Person** data structures (i.e., objects), which each have **score** and **name** attributes.

Unordered List Version	Array Version

highestScore ← 0
highestName ← "unknown"
for each **person** in the **people** list {
    if (**person.**score > **highestScore**) **then** {
        highestScore ← **person.**score
        highestName ← **person.**name
    }
}
print **highestName**
	highestScore ← 0
	highestName ← "unknown"
	for index **i** from **0** to **people.**length-1 {
	    person ← **people[i]**
	    if (**person.**score > **highestScore**) **then** {
	        highestScore ← **person.**score
	        highestName ← **person.**name
	    }
	}
	print **highestName**

Notice that the code is almost identical again, except that we require the index. In order to keep the code simpler, we added a line to set a **person** variable to the element of the array at position **i**. This variable was not necessary. We could have substituted **people[i]** for **person** as follows:

```
for index i from 0 to people.length-1 {
    if (people[i].score > highestScore) then {
        highestScore ← people[i].score
        highestName ← people[i].name
    }
}
```

The point is that each time we want to refer to a different person, we need to supply the index.

Example:

Now assuming that high scores can be tied, we wrote an algorithm that returned a new unordered list of all people who obtained that high score:

```
Algorithm: HighestScore
        people:                 the list (e.g., array) to search through

1.      highestScore ← 0
2.      highestList ← new List
3.      for each person in the people list {
4.          if (person.score > highestScore) then {
5.              highestScore ← person.score
6.              highestList ← new List
            }
7.          if (person.score = highestScore) then
8.              add person.name to highestList
        }
9.      print highestList
```

How can we adjust this to use arrays … even creating a new array to represent the solution ? Let us first convert the code into code that assumes that the **people** list is an array:

```
highestScore ← 0
highestList ← new List
for index i from 0 to people.length-1 {
    if (people[i].score > highestScore) then {
        highestScore ← people[i].score
        highestList ← new List
    }
    if (people[i].score = highestScore) then
        add people[i].name to highestList
}
print highestList
```

Now, how do we adjust the code so that **highestList** is an array ? Recall that an array must have its capacity specified when it is created. How big should this answer be ? Think of the "worst-case-scenario" (i.e., the scenario that causes the array to be its largest size).

It is possible that everyone shares the same high score. In that case, the array would need to be the same size as the original **people** list:

```
highestList ← new Array with capacity people.length
```

How do we add people to the **highestList** ? Well, recall that this array should just be a sequence of **Person** data structures/objects. So, to add a person to the array, we need to indicate where in the array it should go … that is … we need to supply an index in the range from **0** to **highestList.length – 1**.

Logically, the first person added should go at position 0 in the array, the next one in position 1, the third in position 2, etc… So we will need to keep track of the next available location within the array. This can be done with a simple **nextLocation** integer counter which starts at 0.

Then, we simply insert into the array by simply assigning the person to the location in the array specified by **nextLocation**. Here is the code so far:

```
highestScore ← 0
highestList ← new Array with capacity people.length
nextLocation ← 0
for index i from 0 to people.length-1 {
    if (people[i].score > highestScore) then {
        highestScore ← people[i].score
        highestList ← new Array with capacity people.length
        nextLocation ← 0
    }
    if (people[i].score = highestScore) then {
        highestList[nextLocation] ← people[i].name
        nextLocation ← nextLocation + 1
    }
}
print highestList
```

Notice that we need to increase the **nextLocation** counter after we add a new person so that the next person will be added right after it. Do you know what would happen if we did not increase this counter ?

Assume that we were keeping track of the equal highest scores and suddenly we encounter a better high score. What do we do ? All of the high scores that we have added to the **highestList** are no longer highest scores.

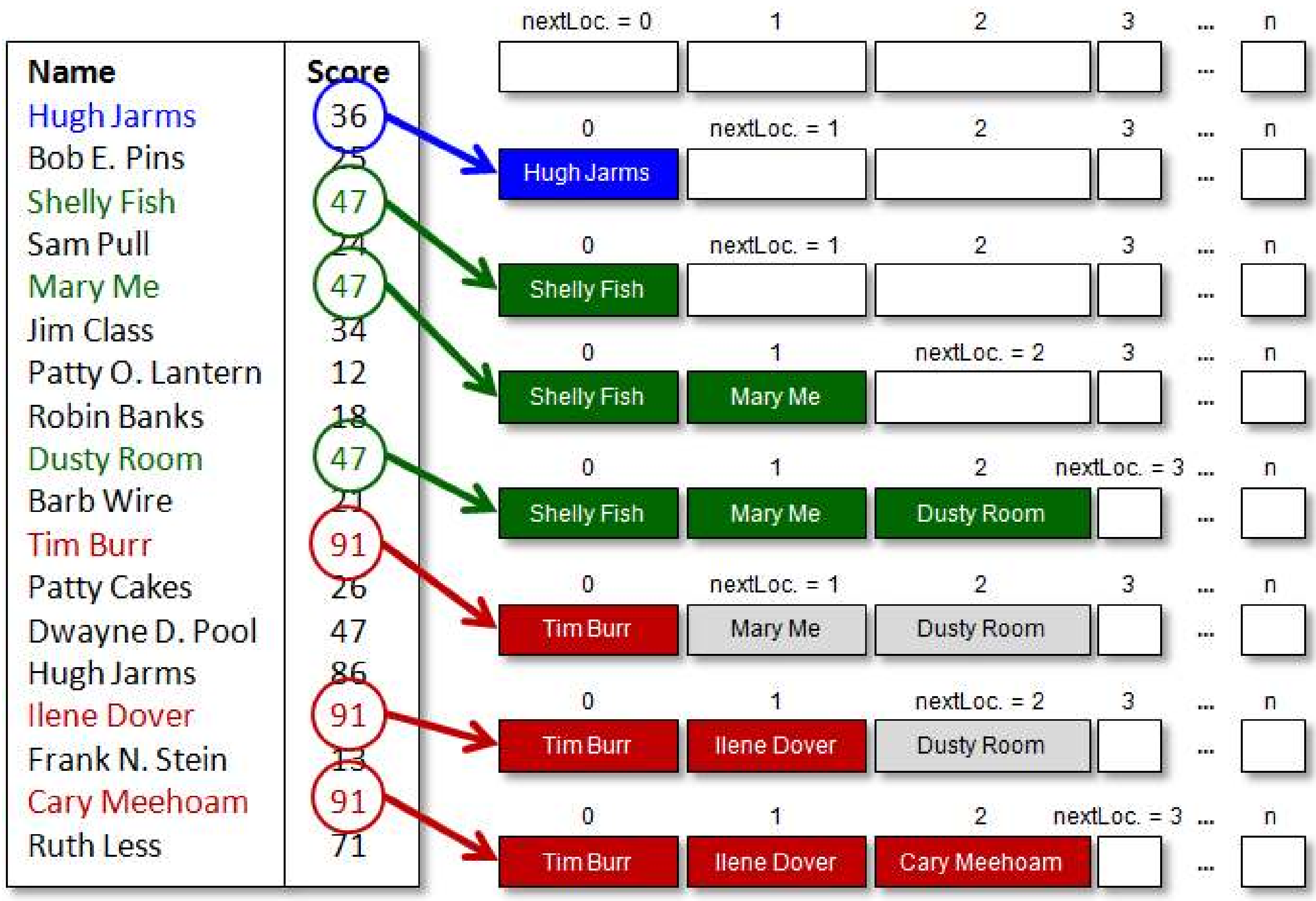
Currently, our code simply replaces the **highestList** with a brand new **highestList**. Where does the previous **highestList** go ? Well, it depends on the programming language. In Processing/JAVA, this old/garbage array will be "garbage collected". That is, the memory space that this array is using up will automatically be discarded at a later point in time by the system. However, in a language like C where WE are responsible for allocating and freeing up the memory, we must make sure to free this memory before re-assigning a new array to the **highestList** variable.

Regardless of the programming language, in both cases we would need to reset the **nextLocation** back to **0** so that new items begin to be placed at the start of the array again.

Interestingly, what would happen if we removed this line from the first **if** statement:

```
highestList ← new Array with capacity people.length
```

Well, the **highestList** would contain some high scores that are no longer valid. However, since we are resetting the **nextLocation** back to zero, all newly encountered high scores will be added to the array such that they overwrite the previous high scores. Notice in this example how the resetting of the **nextLocation** causes overwrite:



As a result, the final array may have some invalid data for all indices greater than **nextLocation**. The **nextLocation** variable actually indicates (at all times) the number of highest scores. Hence, even though the array is of size **n**, if the **nextLocation** variable has the value of **3** at the end of the loop iteration, then there are only **3** valid high scores which are located at the beginning of the array.

Example:

In the previous example, we produced a solution in which the resulting array had a size likely greater than was necessary. For example, if the **people** array had 100 people in it and only two had the highest score, the resulting array would be of size 100, yet only two values in the array would be valid. This is wasteful. How can we adjust the algorithm so that the array has a length that is exactly equal to the number of people with the highest score ?

Logically, we cannot know how many people have the highest score until after we have checked all of the people. So, we must traverse the list at least one time before knowing how big to make the array.

If we leave our code as it is now, before returning we could create a new array with the proper size (i.e., size = **nextLocation**) and then copy the items over from the big array to the other array. Then we can free the memory from the other array if necessary. We just need to append this to the end of our algorithm's code:

```
finalAnswer ← new Array with capacity nextLocation
for index i from 0 to nextLocation-1 {
        finalAnswer[i] ← highestList[i]
}
```

This code will copy the people from one array to the other. The **highestList** array will then be discarded. This is done either through automatic garbage collection (as in Processing/Java), or by manually freeing up the memory (as in C) … depending on the programming language used.

Of course, the above solution may be considered wasteful in that a potentially large array needs to be allocated and then disposed of. This is not a space-efficient solution. Alternatively, we could adjust our algorithm to follow these steps:

1. Find the highest score
2. Find out how many people have the highest score
3. Create an array of the correct size
4. Fill up the array with all those with the highest score.

Separately, steps 1, 2 and 4 each require an iteration through the people in the list:

```
highestScore ← 0                                        // STEP 1 above
for index i from 0 to people.length-1 {
        if (people[i].score > highestScore) then
              highestScore ← people[i].score
}
highestCount ← 0                                        // STEP 2 above
for index i from 0 to people.length-1 {
        if (people[i].score = highestScore) then
              highestCount ← highestCount + 1
}
highestList ← new Array with capacity highestCount      // STEP 3 above
nextLocation ← 0                                        // STEP 4 above
for index i from 0 to people.length-1 {
        if (people[i].score = highestScore) then {
              highestList[nextLocation] ← people[i].name
              nextLocation ← nextLocation + 1
        }
}
print highestList
```

While this solution is more space-efficient, it is less efficient in terms of running time, as it requires an additional iteration through the list. However, with some careful thought, we can combine steps 1 and 2. Do you know how ?

```
highestScore ← 0                                          // STEP 1 & 2 above
highestScoreCount ← 0
for index i from 0 to people.length-1 {
        if (people[i].score > highestScore) then {
              highestScore ← people[i].score
              highestScoreCount ← 0
        }
        if (people[i].score = highestScore) then
              highestScoreCount ← highestScoreCount + 1
}
highestList ← new Array with capacity highestScoreCount   // STEP 3 above
nextLocation ← 0                                          // STEP 4 above
for index i from 0 to people.length-1 {
        if (people[i].score = highestScore) then {
              highestList[nextLocation] ← people[i].name
              nextLocation ← nextLocation + 1
        }
}
print highestList
```

Notice that we just needed to keep track of the number of people with the current high score … the same way as **nextLocation** was used in the previous example to store the number of people with the current highest score.

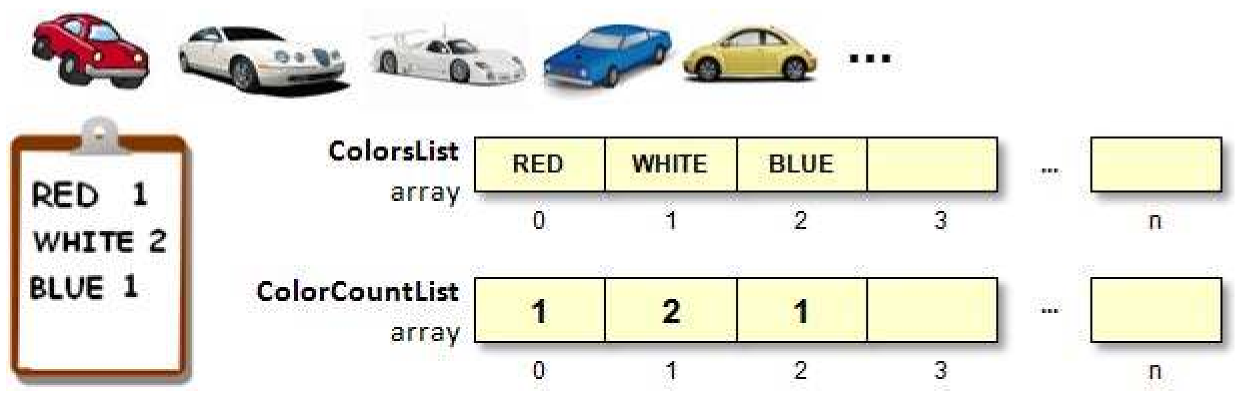
Example:

Recall our example for finding the most popular car color at the autoshow:

```
Algorithm: MostPopularColor
        cars:                   the list to search through

1.      colorsList ← new List
2.      colorCountList ← new List
3.      colorsCount ← CarColorCount()
4.      bestCountSoFar ← 0
5.      bestColorSoFar ← unknown
6.      for each color in the colorsList {
7.            count ← count in colorCountList corresponding to color
8.            if (count > bestCountSoFar) then {
9.                  bestCountSoFar ← count
10.                 bestColorSoFar ← color
              }
        }
11.     print bestColorSoFar
```

```
CarColorCount()
1.      for each car in the cars list {
2.          if (car.color is not in the colorsList) then
3.              add car.color to colorsList
4.          add 1 to colorCountList for the color corresponding to car.color
        }
5.      return the size of colorsList
```

Assume that the **colorsList** and **colorCountList** are now arrays:



The main algorithm remains largely similar, assuming that **CarColorCount()** returns the number of unique colors:

```
Algorithm: MostPopularColor
        cars:                   the list to search through

1.      colorsList ← new array with capacity cars.length
2.      colorCountList ← new array with capacity cars.length
3.      colorsCount ← CarColorCount()
4.      bestCountSoFar ← 0
5.      bestColorSoFar ← unknown
6.      for index i from 0 to colorsCount {
7.          if (colorCountList[i] > bestCountSoFar) then {
8.              bestCountSoFar ← colorCountList[i]
9.              bestColorSoFar ← colorsList[i]
            }
        }
10.     print bestColorSoFar
```

How though can we adjust the **CarColorCount()** function to use arrays ? The code for checking whether a car is in the list or not, is not as simple as this:

```
if (car.color is not in the colorsList) then …
```

We are going to have to go through the whole **colorsList** to determine if the car's color is already in there. This requires an additional loop (see steps 2 though 5 below) with some sort of boolean flag as follows:

```
CarColorCount()

1.    for each car in the cars list {
2.        found ← false
3.        for each color in the colorsList {
4.            if (car.color = color) then
5.                found ← true
          }
6.        if (not found) then
7.            add car.color to the colorsList
8.        add 1 to colorCountList for the color corresponding to car.color
      }
9.    return the size of colorsList
```

This code will add the color to the **colorList** when **found** has not be changed to **true**. The list will therefore contain the proper colors when completed.

But we should adjust the code so that the **colorsList**'s colors are accessed properly by integer indices:

```
CarColorCount()

1.    for each car in the cars list {
2.        found ← false
4.        for index j from 0 to colorsList.length-1 {
5.            if (car.color = colorsList[j]) then
5.                found ← true
          }
6.        if (not found) then
7.            add car.color to the colorsList
8.        add 1 to colorCountList for the color corresponding to car.color
      }
9.    return the size of colorsList
```

But there is a problem on line **7**. Where in the **colorsList** do we add the newly found color ? Logically, we should add it beside the last color that we added. But nowhere do we keep track of *where* the last color was added. So, we should maintain a variable that indicates where to add the next color. This variable will also serve as a counter that indicates how many colors are currently in the list. That is good, because line 9 requires us to return that value:

```
CarColorCount()

1.    colorsCount ← 0
2.    for each car in the cars list {
3.        found ← false
4.        for index j from 0 to colorsCount {
5.            if (car.color = colorsList[j]) then
6.                found ← true
          }
7.        if (not found) then {
8.            colorsList[colorsCount] ← car.color
9.            colorsCount ← colorsCount + 1
          }
10.       add 1 to colorCountList for the color corresponding to car.color
      }
11.   return colorsCount
```

Notice how the loop in line **4** needs only to go from **0** to the number of colors currently in the list (i.e., **colorsCount**), not its capacity.

We are almost done. Line 10 is the last line to fix. In order to increase the count in the **colorsCountList** we need to know the position of **car.color** in the **colorsList**. So we need to, not only determine whether or not a color is *already in* the list, but also *where* (i.e., its index) it is in the list.
So the boolean **found** flag will need to be an integer index instead:

```
CarColorCount()

1.    colorsCount ← 0
2.    for each car in the cars list {
3.        colorPosition ← -1
4.        for index j from 0 to colorsList.length-1 {
5.            if (car.color = colorsList[j]) then
6.                colorPosition ← j
          }
7.        if (colorPosition = -1) then {
8.            colorsList[colorsCount] ← car.color
9.            colorPosition ← colorsCount
10.           colorsCount ← colorsCount + 1
11.           colorCountList[colorPosition] ← 0
          }
12.       colorCountList[colorPosition] ← colorCountList[colorPosition] + 1
      }
13.   return colorsCount
```

Notice how **-1** is used to identify an invalid **colorPosition**. Also, notice how line **11** places a **0** in the **colorCountList** for all newly found colors. On line **12**, both new and repeated colors have their count incremented. That's it. The code is complete.

Example:

Consider a program that continually asks for integers from the user, and adds them to an array as they come in until **-1** is entered, and then does something interesting with the numbers that were entered. What is the structure of the algorithm ?

Algorithm: GetValidNumbers

```
1.      numbers ← new array with some initial capacity
2.      count ← 0
3.      entered ← get a number from the user
4.      while (entered is not -1) {
5.          numbers[count] ← entered
6.          count ← count + 1
7.          entered ← get a number from the user
        }
8.      do something with numbers
```

The code is straight forward. However, can you foresee a problem that may arise ? Notice that each number coming in is added to the array according to the **count** position, which is incremented each time a valid number arrives. When the loop has completed, **count** represents the number of integers that were entered and added to the array.

It is possible that we may attempt to add an item beyond the array's capacity. In such a case, the program will usually crash (or stop unexpectedly and non-gracefully).

How do we address this potentially serious problem ? Since arrays are fixed size, we cannot simply make more room within the existing array. Rather, a new *bigger* array must be created and all elements must be copied into the new array. But how much bigger ? It's up to us.

Here is how we could change the code to increase the array size by 5 each time it gets full …

```
Algorithm: GetValidNumbers

1.    numbers ← new array with some initial capacity
2.    count ← 0
3.    entered ← get a number from the user
4.    while (entered is not -1) {
5.        if (count >= numbers.length) then {
6.            tempArray ← new array with capacity numbers.length + 5
7.            for index i from 0 to numbers.length-1 {
8.                tempArray[i]  ← numbers[i]
              }
9.            numbers ← tempArray
          }
10.       numbers[count] ← entered
11.       count ← count + 1
12.       enteredNumber ← get a number from the user
      }
13.   do something with numbers
```

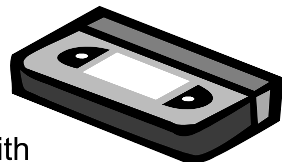
Of course, we can increase by any amount each time, perhaps even doubling the size.

Example:

Assume that we are given an array of Person objects and we need to discard (or remove from the array) all people below the age of 18.

```
Algorithm: RemoveYoungsters
      people:          the list to search through

1.    for each person in the people list {
2.        if (person.age < 18) then
3.            remove person from the people list
      }
```

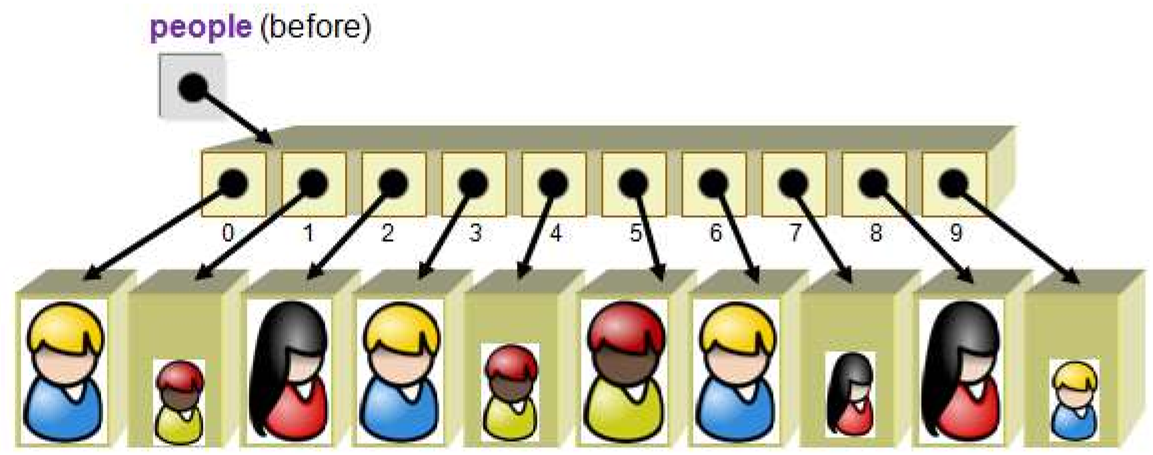
How could we do this using an array ? Well, an array is fixed size, so when we remove data, the array will not get smaller. The situation is analogous to a program recorded on a videotape, we cannot actually remove the item but we can overwrite it (i.e., replace it) with a new value.

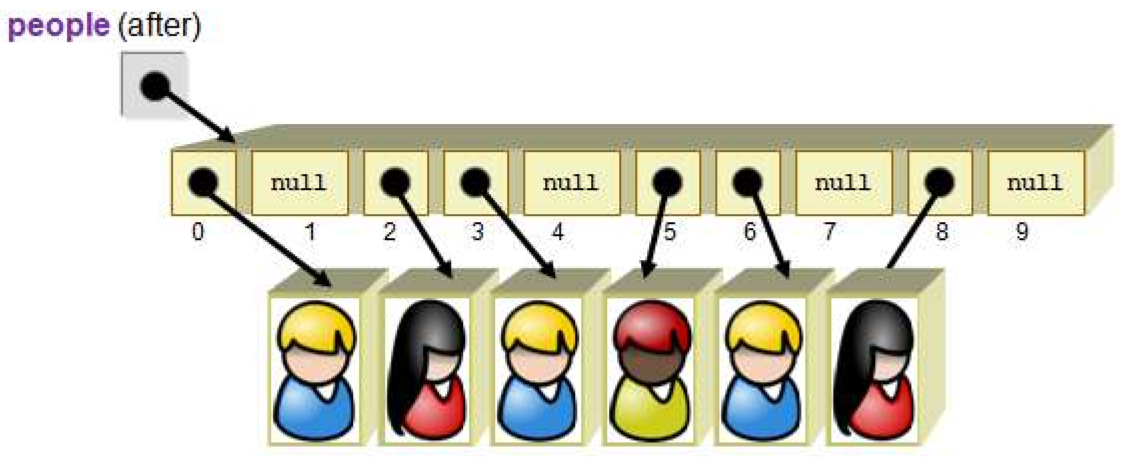


So, to delete a piece of information from an array, you usually replace it with **0** or **null**. The array will stay the same size, but the data will be deleted.

```
Algorithm: RemoveYoungstersArrayVersion
        people:         the array to search through

1.      for index i from 0 to people.length-1 {
2.          if (people[i].age < 18) then
3.              people[i] ← null
        }
```

This will replace all underage people with **null**, but the result array will be filled with "gaps" or holes:





The holes may present two problems. First, we don't know how many people are left in the array. Second, when looping through the array we will encounter **null** objects that we need to deal with properly.

To handle the issue regarding the amount of valid data remaining in the array, we can always maintain a variable indicating the number of such valid elements as follows:

```
Algorithm: RemoveYoungstersArrayVersion2
    people:         the array to search through

1.  count ← 0
2.  for index i from 0 to people.length-1 {
3.      if (people[i].age < 18) then
4.          people[i] ← null
5.      otherwise
6.          count ← count + 1
    }
```

This **count** variable will indicate the number of people who are 18 or over. The code above assumes that the array was filled with people.

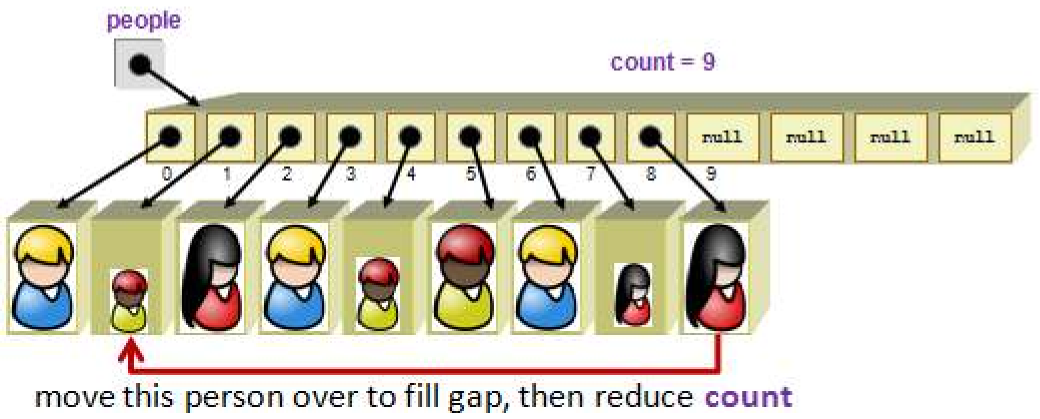
The second issue of being able to handle the holes implies that we check for a hole each time:

```
Algorithm: RemoveYoungstersArrayVersion3
    people:         the array to search through

1.  count ← 0
2.  for index i from 0 to people.length-1 {
3.      if (people[i] is not null) then {
4.          if (people[i].age < 18) then
5.              people[i] ← null
6.          otherwise
7.              count ← count + 1
        }
    }
```

However, it is often unpleasant to continually check for "holes" in the array like this. In fact, in a typical application we are likely more often going to need to traverse through elements of an array for display purposes than to search through the array to remove data. Therefore, it would be more advantageous to "fill-in" the hole each time so that the valid array elements are at the front-most part of the array at all times.

Assume that we have such a valid array in which all the elements are at the front-most part of the array. Assume then, that we always have a count as to how many people are stored in the array at all times.

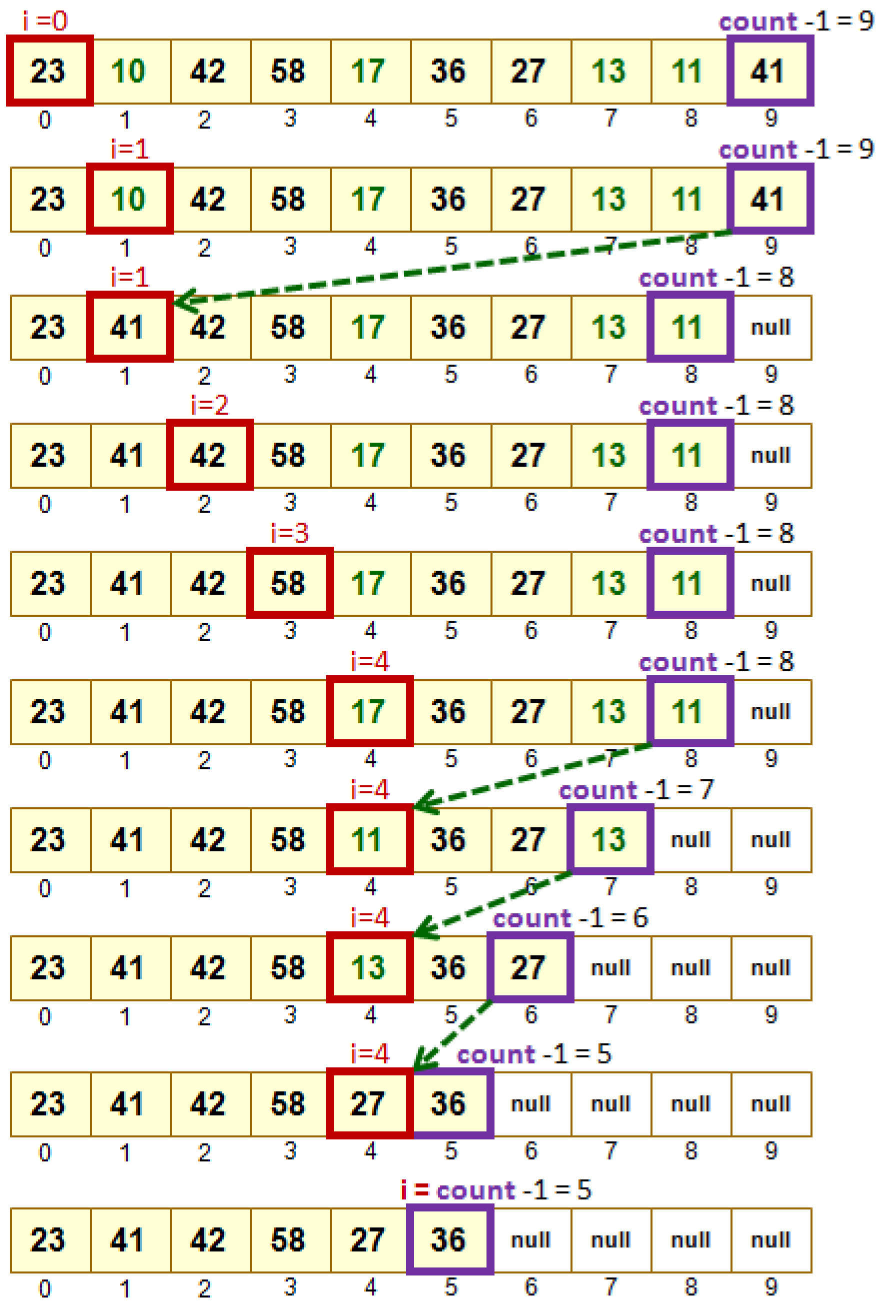
Whenever we decide to remove a person, we can grab the person from the end of the array and place it in that spot, then reduce our counter by one to indicate that we have one less piece of data. We can even place **null** in the last location of the array to erase the data that was there.



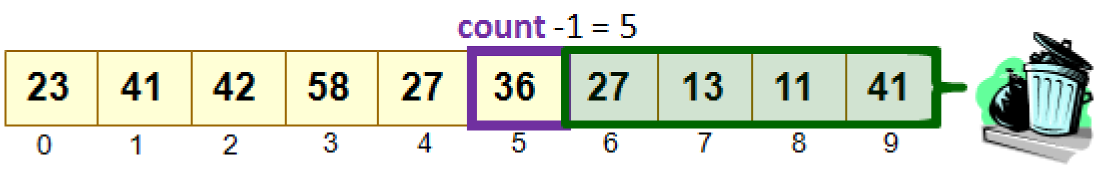
However, it is possible that the last person in the array that we try to move over is also underage. Therefore, we could go ahead and move it over, but make sure to check the age of the person that we moved over as well before we move on to the next index in the array. Here is the code:

```
Algorithm: RemoveYoungstersArrayVersion4
        people:         the array to search through
        count:          number of people in the array

1.      for index i from 0 to count – 1 {
2.              if (people[i].age < 18) then {
3.                      people[i] ← people[count – 1]
4.                      people[count – 1] ← null
5.                      count ← count - 1
6.                      i ← i – 1      // decrease index so that next time we check this new item
                }
        }
```

Here is a picture of what is happening (the ages are shown in the array instead of the person in order to save space):

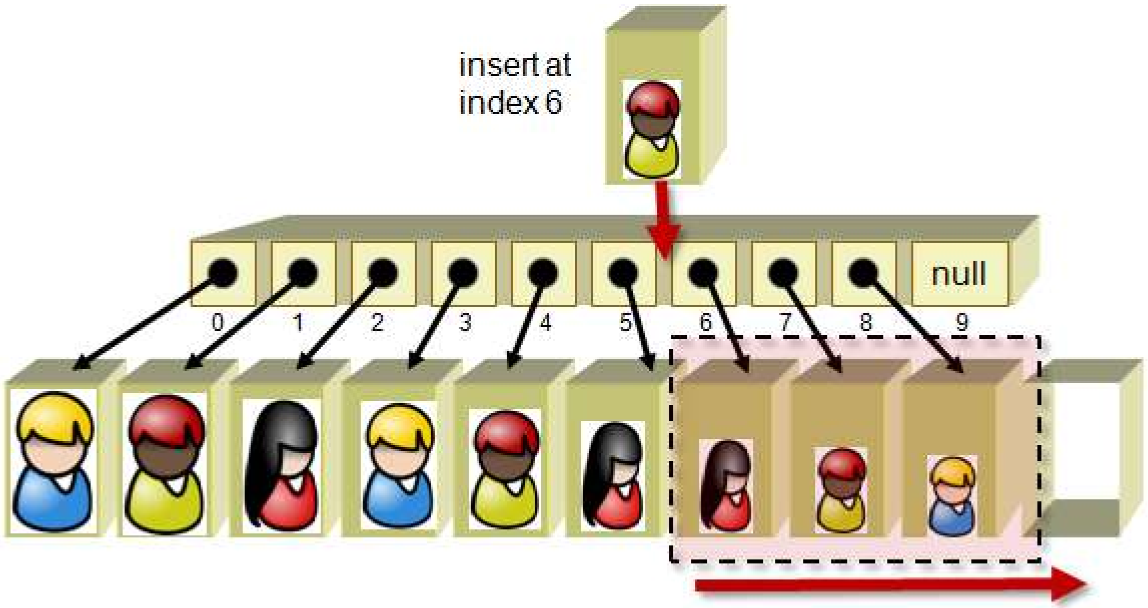


Notice how the count continually decreases as elements are removed from the array. It is not actually necessary to ***move*** the item to the open "holes". The item may simply be ***copied*** over, leaving "garbage" at the end of the array:

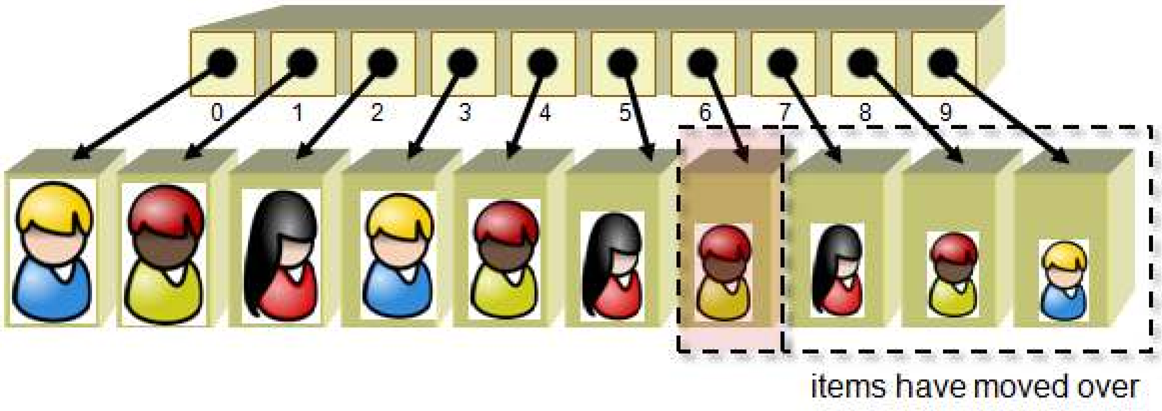


Example:

Assume that we already had a list of people sorted by their ages. How can we write code that will ***insert*** a new person into the appropriate location in the array so that it remains sorted ?



To insert at some index position (e.g., **i**) we must shift all array items from indices **i** onwards one position further in the array:



Here is the code to do this:

```
Algorithm: ArrayInsert
        people:         the array of sorted people
        newPerson:      the new person to insert into the array
        count:          number of items currently in the array

1.      if (count < people.length) then {
2.          index ← 0
3.          while (index < count) {
4.              if (people[index].age < newPerson.age) then {
5.                  for index i from count-1 down to index {   // count backwards
6.                      people[i+1]  ← people[i]
                    }
7.                  people[index]  ← newPerson
8.                  count  ← count + 1
9.                  index ← count       // Need this in order to quit
                }
10.             index ← index + 1       // Need this to check next position
            }
        }
```

Notice how the search for the insertion position continues until a person in the array has an age less than that of the person being inserted. Then every person after that "found" position is moved over to the right and the new person is simply added to that newly found position.

5.4 Efficient Searching Using Arrays

We have discussed linear searching and how it sometimes requires us to iterate through all the items in a list whereas at other times we can stop the iteration once we have found something that positively answers our problem at hand. In the worst case, however, it can require a full search through all the items.

There is another very popular search strategy called a binary search:

> A ***binary search*** is an efficient method for finding a particular value in a sorted list, that consists of continually reducing the search space by half during each search step.

As stated, in order to perform a binary search it is necessary for the elements to be in sorted order. If the elements are out of order, the algorithm will not work.

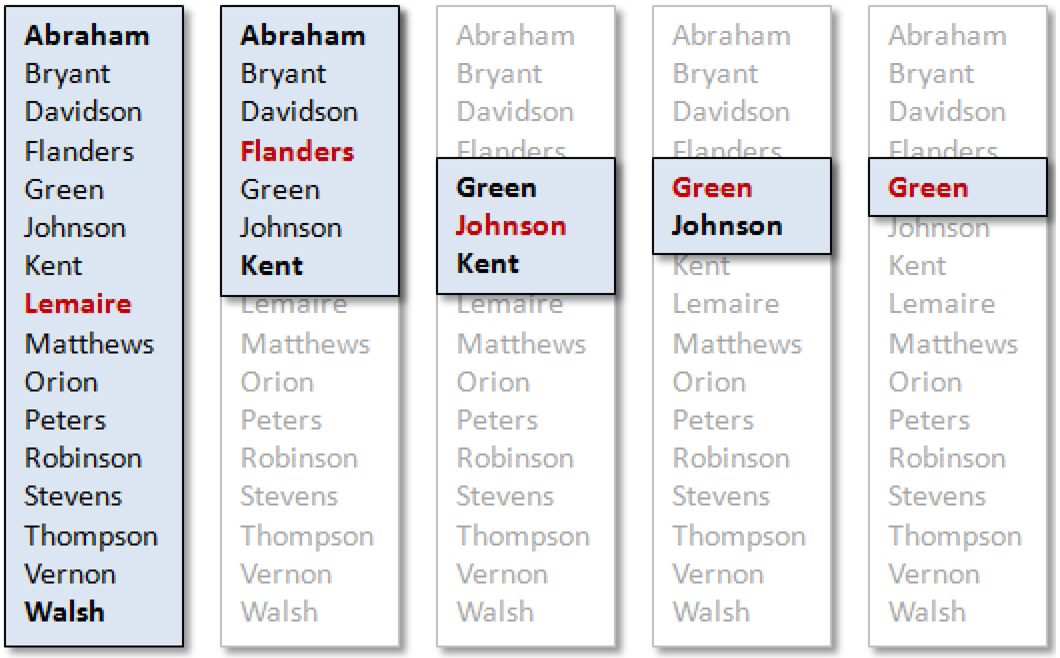
The idea behind a binary search is easily understood using the analogy of looking up someone's name in the white pages of a phone book. Image that you are looking for the last name

“Peterson”. If you opened the book in the middle and looked at the top of the page, chances are that you would see a name that comes alphabetically before Peterson (e.g., Lanthier”) or alphabetically after it (e.g., “Ryans”). If you saw “Ryans”, for example, then you know that “Peterson” comes before it in the book (since the names are sorted), so you don’t bother checking the 2nd half of the book, but instead turn your attention to the first half of the book. Then, the problem kinda starts all over again with the first half of the book in that you will now check from book from “A” to “Ryans”, ignoring the portion from “Ryans” to “Z”. The process continues until the name is found, each time a big chunk of the phone book is eliminated from the potential pages to search.

The “binary” from “binary search” indicates that the data is repeatedly divided into two halves. How do we find the half way point ? Well, if the first page is **1** and the last page is **2000**, we just do **(2000 – 1) / 2 = 999.5** which can be rounded up to **1000**. But what about when we are searching the top half of the book … perhaps from page **1000** to **2000** ? Then the same formula of **(2000 – 1000) / 2 = 500** which is not the middle. **500** is actually the number of pages from page **1000** to the middle of the **1000** to **2000** range. So we need to add the offset of the start page. The formula is therefore:

midPage = startPage + (endPage – startPage) / 2
or
midPage = (startPage + endPage) / 2

Here is an example of how the algorithm works if we were looking for the name **Green**:



Here is a more formal algorithm:

```
Algorithm: SearchPhoneBook
        phonebook:              the book to search through
        searchName:             the name you are looking for

1.      startPage ← 1
2.      endPage ← 2000          // or whatever the last page is
3.      found ← false
4.      while ((not found) and (startPage is not greater than endPage)) {
5.              midPage ← (startPage + endPage) / 2
6.              open the phonebook at midPage
7.              currentName ← name at the top of midPage
8.              if (currentName is the same as searchName) then
9.                      found ← true
10.             else {
11.                     if (currentName comes alphabetically before searchName) then
12.                             startPage ← midPage + 1
13.                     else
14.                             endPage ← midPage - 1
                }
        }
```

Notice how the **startPage** (or **endPage**) is adjusted in line **12** (or **14**) to be one more (or less) than the **midPage**. That is because we have already checked the **midPage**, so there is no need to have it remain in the list of items to search.

Of course, this algorithm works for any collection of sorted items. Assume that the items are stored in an array. We can make the algorithm even more formal and compact as shown below. This pseudocode assumes that the items in the array can be compared using a **<** operator:

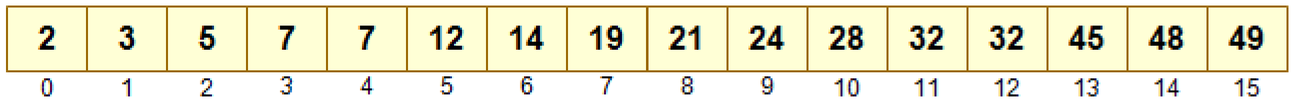
```
Algorithm: BinarySearch
        items:          the array to search through
        searchItem:     the item you are looking for

1.      s ← 0
2.      e ← items.length - 1
3.      location ← -1
4.      while ((location < 0) and (s <= e)) {
5.              m ← (s + e) / 2
6.              if (items[m] = searchItem) then
7.                      location ← m
8.              else {
9.                      if (items[m] < searchItem) then
10.                             s ← m + 1
11.                     else
12.                             e ← m - 1
                }
        }
13.     print location
```

The code has been adjusted to return the **location** of the found item, not just a boolean showing whether or not it was found. Of course, if the item is not found, it will return **-1**.

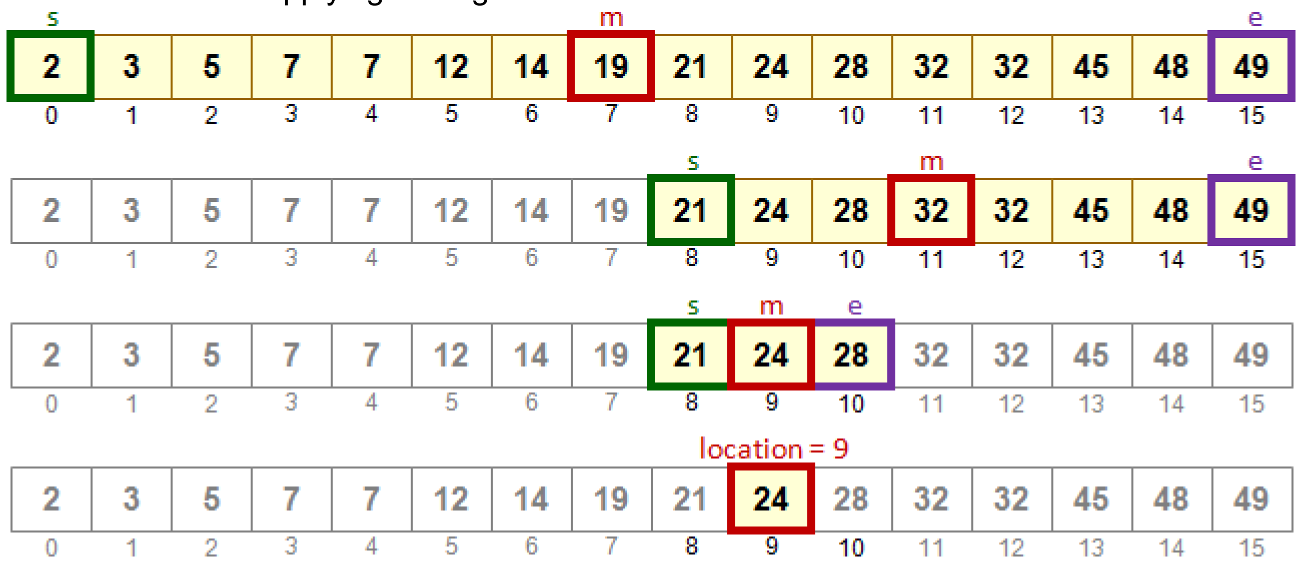
Notice that it loops through the items one at a time and returns the same result. However, this code is simpler than the binary search. In order to see the advantage of the binary search, let us look at a real example.

Example:

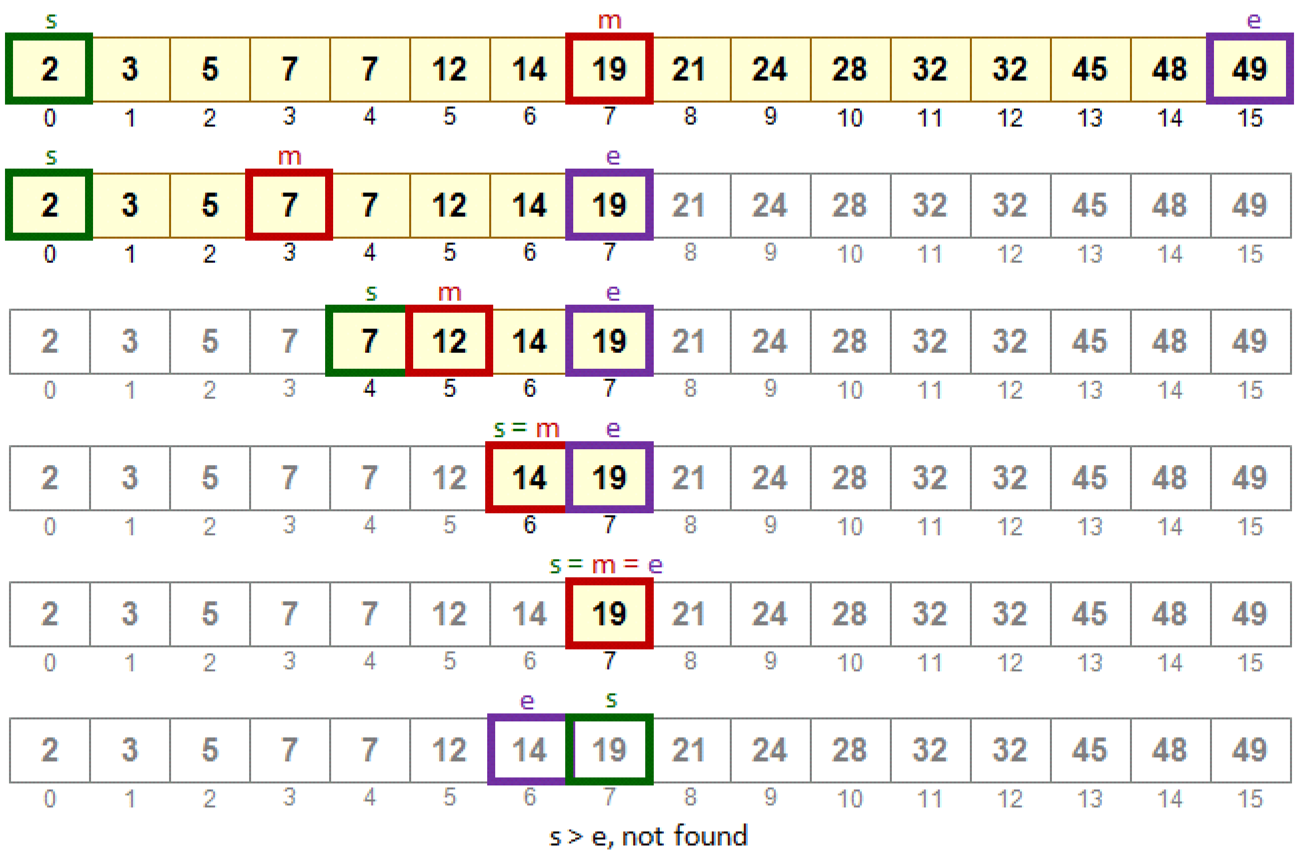
As an example, assume that we have the following array of numbers and that we want to determine whether or not the number **24** lies within the array:



Here is the result of applying the algorithm:



… and here is the result if we were searching for the number **18**:



You may feel that this seems more complicated than simply searching through the elements one-by-one. Yes, it is more complicated. Here is a comparative linear search:

```
Algorithm: LinearSearch
        items:          the array to search through
        searchItem:     the item you are looking for

1.      i ← 0
3.      location ← -1
4.      while ((location < 0) and (i <= items.length - 1)) {
5.          i ← i + 1
6.          if (items[i] = searchItem) then
7.              location ← i
        }
13.     print location
```

But think about the advantages. Imagine checking the phone book for a name by always starting your search from the front of the book and checking page by page. Do you understand the advantage yet ?

In our above example, a linear search for the number 24 would require us to examine 10 array items (i.e., 2, 3, 5, 7, 7, 12, 14, 19, 21, 24), while the binary search required us to examine only 3 array items (i.e., 19, 32, 24) before the number was found.

When searching for a number that is not there (e.g. 20), the linear search would have to be exhaustive, checking all 16 items. However, the binary search required only 5 to be examined.

Of course, the binary search code is longer and has more variables, so there is a bit of "overhead" in getting it to work. Hence, the linear search is often much better to use when a small number of items need to be searched. The advantage of the binary search comes when the array size is large. For example, in the worst case scenario an array of around **1,000,000** items would require us to search all **1,000,000** items if the item is not in the array. However, a binary search would only require a search of around **$\log_2$(1,000,000) ~= 20** items !!! The logarithm (base 2) comes into play because we are continually discarding ½ of the array items during each round through the loop. Can you imagine the time-savings of searching 1GB of data ? It would check only **30** items at most in place of **1,000,000,000** !!!

As you continue with your degree in computer science, you will learn much more about efficiency.

5.5 Arrays in Processing

Recall that in Processing & Java, variables are defined by specifying their type (primitive or object) as well as a label (the variable's name). Array variables are defined similarly, but with square **[ ]** brackets. Below is a table showing how to declare variables that hold a single value verses variables that hold arrays (or multiple values):

Single-value variables	Array variables

`boolean hungry;`
`int days;`
`char gender;`
`float amount;`
`double weight;`
 	
`Car myCar;`
`FullName myName;`
`Address myHomeAddress;`
`BankAccount myAccount;`
	`boolean[] hungry;`
	`int[] days;`
	`char[] gender;`
	`float[] amount;`
	`double[] weight;`
	
	`Car[] myCar;`
	`FullName[] myName;`
	`Address[] myHomeAddress;`
	`BankAccount[] myAccount;`

So you can see that arrays are declared similarly to regular variables. Note that the square brackets may appear either with the type (as shown) or with the variable's name as follows:

```
int   days[];
Car   myCar[];
```

Now these array variable declarations simply reserve space to store array objects, but it actually does not create the array object.

So, the following code would print out **null** because the variable is not yet initialized:

```
    int[]   days;
    println(days);
```

Notice above that we did **not** use the square brackets **[]** when you are ***using*** the array variable in your code ... you only use the brackets when we ***define*** the variable.

To ***use*** this variable (which is just like any other variable), we must give it a value. What kind of value should it have ? An array of course. But it may surprise you to know that we do not call a constructor to create arrays. Instead, we use a special (similar) syntax as follows:

```
    new ArrayType[ArraySize]
```

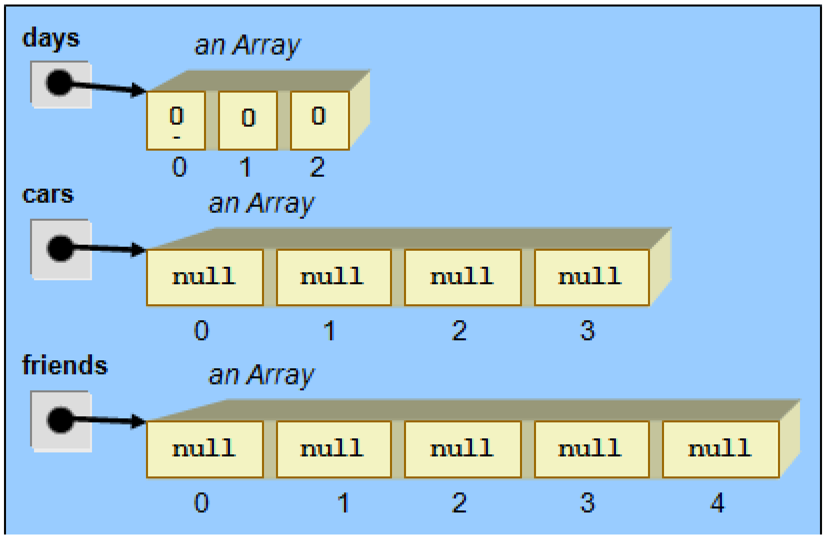
This is the template to create an array that can hold up to `ArraySize` values of type `ArrayType`. Remember that arrays are fixed size, so you cannot enlarge them later. If you are unsure how many items that you want to put into the array, you should chose an over-estimate (i.e., a maximum bound) for its size. Here are some examples of how to give a value to our 6 array variables by creating some fixed-size arrays:

```
int[]          days;
double[]       weights;
String[]       names;
Car[]          rentals;
Person[]       friends;

days = new int[30];       // creates array that can hold 30 ints
weights = new double[100];// creates array that can hold 100 doubles
names = new String[3];    // creates array that can hold 3 String objects
rentals = new Car[500];   // creates array that can hold 500 Car objects
friends = new Person[50]; // creates array that can hold 50 Person objects
```

Once we create arrays using the code above, the arrays themselves simply reserve enough space to hold the number of objects (or data types) that you specified. However, it does not create any of those objects!! The array is NOT initialized with new objects in each location. Newly created arrays are filled with 0 for numbered arrays, character 0 (i.e., the **null** character) for **char** arrays, **false** for **boolean** arrays and **null** for Object-type arrays.

Hence, the following code produces the result shown in the picture here:

```
int[]      days;
Car[]      cars;
Person[]   friends;

days = new int[3];
cars = new Car[4];
friends = new Person[5];
```



Notice that the arrays which hold object types are simply filled with **null** values … there are no **Car** objects nor **Person** objects created in the above code.

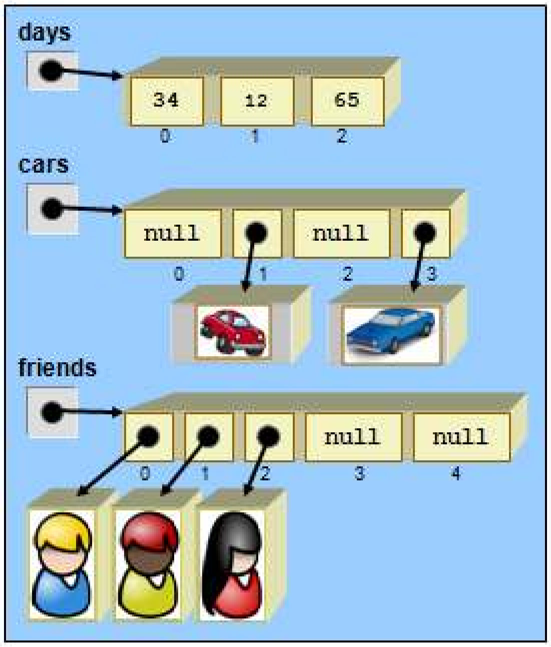
At any time, if we would like to ask an array how big it is (i.e., its size or capacity), we can access one of its special attributes called **length** as follows:

```
println(days.length);     // displays 3
println(cars.length);     // displays 4
```

Remember that the **length** of an array is its overall capacity, it is not the number of elements that you put into the array.

Values are actually assigned to an array using the **=** operator just as with other variables. Here is an example of how to fill in our arrays:

```
int[]      days;
Car[]      cars;
Person[]   friends;

days = new int[3];
cars = new Car[4];
friends = new Person[5];

days[0] = 34;
days[1] = 12;
days[2] = 65;
cars[1] = new Car("Red");
cars[3] = new Car("Blue");
friends[0] = new Person(...);
friends[1] = new Person(...);
friends[2] = new Person(...);
```



The picture here shows the result from this code →

Notice that we can insert an object at any location in the array, provided that the index is valid.

The following two lines of code would produce an **ArrayIndexOutOfBoundsException**:

```
days[3] = 87;                  // Error: index 3 is out of range
cars[10] = new Car("Yellow");   // Error: index 10 is out of range
```

A very common mistake made when learning to use arrays is to declare the array variable, but forget to create the array and then try to use it.   For example, look at this code:

```
Person[]  friends = new Person[10];
println(friends[0].firstName);   // Error!
println(friends[0].age);         // Error!
println(friends[0].gender);      // Error!
```

Although the above code does *create* an array that can *hold* **Person** objects, it actually never creates any **Person** objects.   Hence, the array is filled with 10 values of **null**.

The code will produce a **NullPointerException** because **friends[0]** is **null** here and we are trying to access the attributes of a **null** object. We are really doing this: **null.firstName** … which makes no sense.

Assigning individual values to an array like this can be quite tedious, especially when the array is large. Sometimes, in fact, we already know the numbers that we want to place in an array (e.g., we are using some fixed table or matrix of data that is pre-defined). In this case, to save on coding time JAVA allows us to assign values to an array at the time that we create it. This is done by using braces **{ }**. In this case, neither the **new** keyword, the **type** nor the **size** of the array are specified. Instead, we supply the values on the same line as the declaration of the variable. Here are some examples:

```
int[]       ages = {34, 12, 45};
double[]    weights = {4.5, 23.6, 84.124, 78.2, 61.5};
boolean[]   retired = {true, false, false, true};
String[]    names = {"Bill","Jennifer","Joe"};
char[]      vowels = {'a', 'e', 'i', 'o', 'u'};
```

Here, the array's size is automatically determined by the number of values that you specify within the braces, each value separated by a comma. So the sizes of the arrays above are 3, 5, 4, 3 and 5, respectively.

Objects may also be created and assigned during this initialization process as follows …

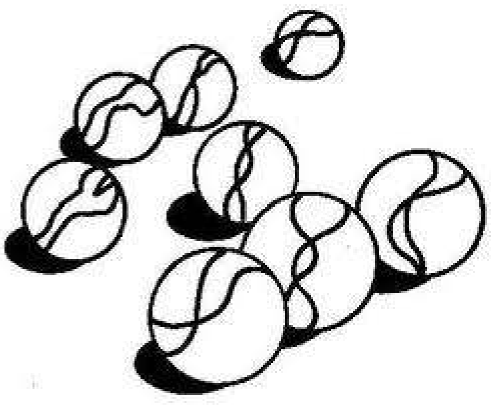
```
Person[]   people = {new Person("Hank", "Urchif", 19, 'M', false),
                     new Person("Don", "Beefordusk", 23, 'M', false),
                     new Person("Holly", "Day", 67, 'F', true),
                     null,
                     null};
```

Here we are actually creating three specific **Person** objects and inserting them into the first three positions in the **people** array. Notice that you can even supply a value of **null**, for example if you wish to leave some extra space at the end for future information. Hence the people array above has a capacity (i.e., length) of **5**.

Example:

Recall our example in which a ball was thrown around the window. How can we adjust the code so that **multiple balls** are able to be thrown around ?

```
final int   RADIUS = 40;
final float ACCELERATION = 0.10;

Ball        b;    // the Ball
boolean     grabbed;

class Ball {      // as defined earlier
      ...
}

void setup()() {
  size(600,600);
  aBall = new Ball(width/2, height/2,
                   random(TWO_PI), 10);
  grabbed = false;
}

void draw() { ... }
void mousePressed() { ... }
void mouseReleased() { ... }
```



To begin, we will need to store an array of balls. We can adjust **b** to be an array **balls**. Then in the **setup()** method, we need to create the array and fill it up with some balls:

One Ball Version	Array of Balls Version

```
Ball        b;    // the Ball
boolean     grabbed;
...
void setup() {
  ...
  aBall = new Ball(width/2, height/2,
                   random(TWO_PI), 10);
  grabbed = false;
}
```
	```
	Ball[]      balls;      // lotsa balls
	boolean     grabbed;
	...
	void setup() {
	  ...
	  balls = new Ball[10];
	  for (int i=0; i<balls.length; i++)
	    balls[i] = new Ball(width/2, height/2,
	                        random(TWO_PI), 10);
	  grabbed = false;
	}
	```

Notice how we need to use a loop to fill in the array at each location. If we did not do this, the array would be filled with **null**. Each ball is made as before, and placed in the array at consecutive locations.

The **draw()** procedure will now need to draw ALL the balls. So we will need to loop through them all in order to ensure that all get displayed. Here is how the code gets adjusted:

One Ball Version

```
void draw() {
  background(0,0,0);
  ellipse(b.x, b.y, 2*RADIUS, 2*RADIUS);

  if (!grabbed) {
    b.x = b.x + int(b.speed * cos(b.direction));
    b.y = b.y + int(b.speed * sin(b.direction));
  }
  else {
    b.x = mouseX;
    b.y = mouseY;
  }

  b.speed = max(0, b.speed - ACCELERATION);

  if ((b.x + RADIUS >= width)||(b.x - RADIUS <= 0))
    b.direction = PI - b.direction;
  if ((b.y + RADIUS >= height)||(b.y – RADIUS <= 0))
    b.direction = - b.direction;
}
```

Array of Balls Version

```
void draw() {
  background(0,0,0);
  for (int i=0; i<balls.length; i++) {
    ellipse(balls[i].x, balls[i].y, 2*RADIUS, 2*RADIUS);

    if (!grabbed) {
      balls[i].x = balls[i].x + int(balls[i].speed * cos(balls[i].direction));
      balls[i].y = balls[i].y + int(balls[i].speed * sin(balls[i].direction));
    }
    else {
      balls[i].x = mouseX;
      balls[i].y = mouseY;
    }

    balls[i].speed = max(0, balls[i].speed - ACCELERATION);

    if ((balls[i].x + RADIUS >= width)||(balls[i].x - RADIUS <= 0))
      balls[i].direction = PI - balls[i].direction;
    if ((balls[i].y + RADIUS >= height)||(balls[i].y - RADIUS <= 0))
      balls[i].direction = - balls[i].direction;
  }
}
```

Notice how the **for** loop wraps around the code. Also, notice how each reference to **b** is simply replaced by **balls[i]** to indicate the particular ball in the array.

However, there is a problem in regards to grabbing the ball. What ball are we grabbing ? Should we be able to grab any ball ? Probably.

We will need to make a change so that instead of remembering ***whether or not we grabbed*** a ball, we should remember ***which ball we grabbed***, if any at all.

Therefore, we will change our **grabbed** boolean to be of type Ball, so that we can keep track of the particular ball that was grabbed:

```
boolean    grabbed;
```
will become
```
Ball  grabbed;
```

In the **setup()** procedure, we will set it to **null**, since we have not grabbed any balls when the program first starts.

```
grabbed = false;
```
will become
```
grabbed = null;
```

How do we change the **if (!grabbed)** line in the **draw()** procedure ? We want to move each ball except the one that was grabbed. So we move the ball at location **i** as long as it is not the grabbed one. So we need to change this line to **if (grabbed != balls[i])** in the **draw()** procedure.

Likewise, we need to adjust the **mousePressed()** procedure to search through all the balls and find out which one was grabbed, then remember it as follows:

One Ball Version

```
void mousePressed() {
  if (dist(b.x, b.y, mouseX, mouseY) < RADIUS)
    grabbed = true;
}
```

Array of Balls Version

```
void mousePressed() {
  for (int i=0; i<balls.length; i++) {
    if (dist(balls[i].x, balls[i].y, mouseX, mouseY) < RADIUS)
      grabbed = balls[i];
  }
}
```

Likewise, we need to adjust the **mouseReleased()** procedure to “un-remember” the grabbed ball as follows:

One Ball Version

```
void mouseReleased() {
  if (grabbed) {
    b.direction = atan2(mouseY - pmouseY, mouseX - pmouseX);
    b.speed = int(dist(mouseX, mouseY, pmouseX, pmouseY));
  }
  grabbed = false;
}
```

Array of Balls Version

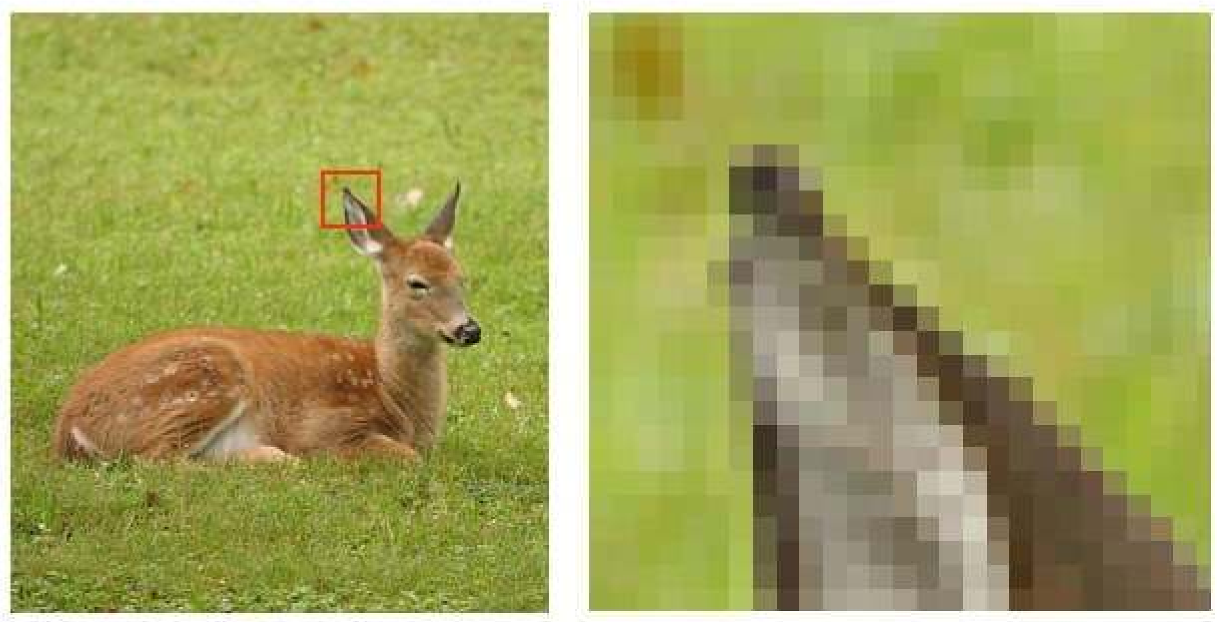
```
void mouseReleased() {
  if (grabbed != null) {
    grabbed.direction = atan2(mouseY - pmouseY, mouseX - pmouseX);
    grabbed.speed = int(dist(mouseX, mouseY, pmouseX, pmouseY));
  }
  grabbed = null;
}
```

Notice how we now use the grabbed variable to set the direction and speed of the grabbed ball according to where we throw it.

5.6 2D Arrays

In all of our search-related examples so far we assumed that we had a list of items that we are searching through. A list of items is considered 1-dimensional in that we travel one way through the list (although this could be done forwards or backwards). Sometimes, however, we need to perform 2-dimensional searches by traveling up, down, left or right through data that is usually arranged in a grid.

One particular class of problems that apply to 2-dimensional data is called ***image processing***. That is, there are many algorithms that process and manipulate images. All images can be stored in pixels arranged in an (x,y) grid:



Sometimes the images are compressed and stored differently (e.g., jpg) but in our examples, we will assume that the data is in what is known as a "raw" format where each pixel (representing a dot of the image) is accessible and modifiable by a unique (x,y) location in the image. Therefore, we can imagine that the image is stored in a 2-dimensional array as rows and columns of pixels. Therefore, we will access a particular pixel in the **image** by using this notation: **image**[**row**][**column**]

Here, each pixel requires two indices…one to represent the row (starting at zero), the other to represent the column (starting at 0 as well). While there are many ways to represent pixels in an image, we will assume for our examples that each pixel value is either stored as a:

- **grayscale** value: Usually a number from 0 to 255 representing the *amount of "gray"* at that location in the image.
- **RGB** value: A set of 3 numbers (each from 0 to 255) representing the *amount of "red", "green" and "blue"* at that location in the image.

Example:

How would we print out the grayscale values of a gray scale image stored in an array with **R** rows and **C** columns ? What does the loop structure look like ?

```
Algorithm: PrintGrayValues
        image:          the 2D array representing the image
        R:              number of rows in the image
        C:              number of columns in the image

1.      for each row r from 0 to R-1 {
2.          for each column c from 0 to C-1 {
3.              print image[r][c]
            }
        }
```

Nested **for** loops are usually everybody's favorite control structure for iterating through the elements of a 2-dimensional array.

The above code assumes that the value of the array is indeed the grayscale value. How would the code differ if the image was stored in RGB format ? This depends on how the data is actually stored. For example, it is possible that, instead of a single grayscale byte value per pixel, the array may actually hold a data structure with accessible r, g, b values:

```
define Pixel to be made of {
        .red
        .green
        .blue
}
```

In this case, we would simply need to access the appropriate values:

```
Algorithm: PrintRGBValues
        image:          the 2D array of Pixel objects representing the image
        R:              number of rows in the image
        C:              number of columns in the image

1.      for each row r from 0 to R-1 {
2.          for each column c from 0 to C-1 {
3.              print image[r][c].red
4.              print image[r][c].green
5.              print image[r][c].blue
            }
        }
```

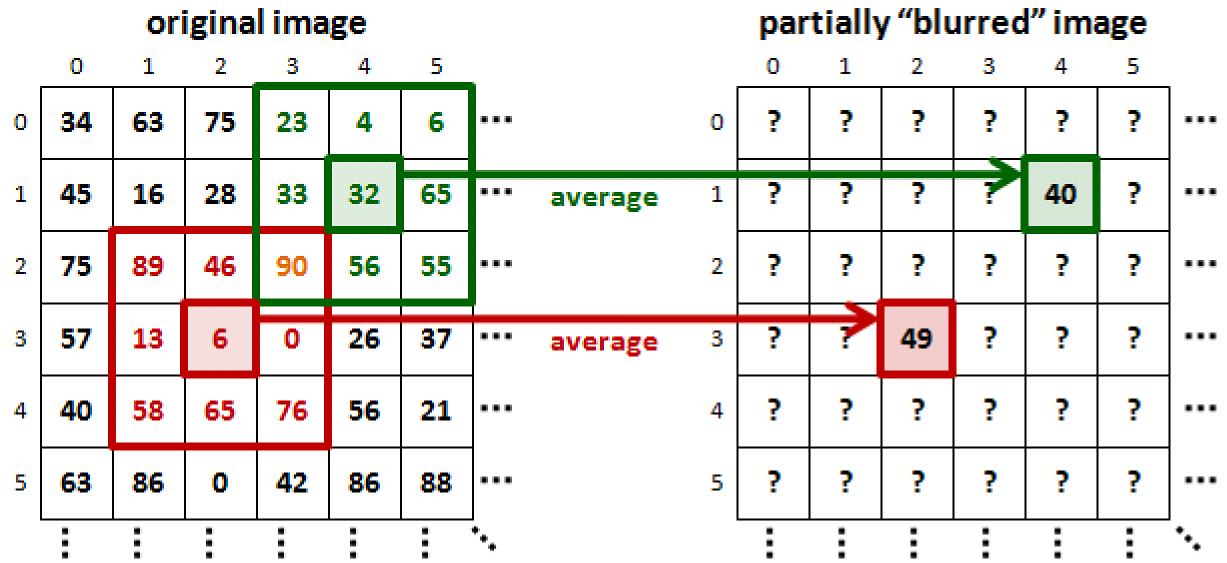
In some programming languages, however, the pixels may be stored as large 3-byte or 4-byte integers that encodes the RGB values in some manner (sometimes including gray or black level as a 4th byte). In this case, the programming language will often provide functions for extracting the relevant data and thus the code may look something like this:

```
Algorithm: PrintRGBValues
        image:          the 2D array containing the image
        R:              number of rows in the image
        C:              number of columns in the image

1.      for each row r from 0 to R-1 {
2.          for each column c from 0 to C-1 {
3.              print redAmount(image[r][c])
4.              print greenAmount(image[r][c])
5.              print blueAmount(image[r][c])
            }
        }
```

Regardless of the exact manner in which the color values are accessed, the processing of the image data occurs in a similar manner.

Example:

A common operation in image processing is that of "blurring" an image. There are multiple ways to blur an image. One way is simply to replace a pixel's value with a new value that corresponds to the average values of the pixels around it (including itself). A simple "box blur" would examine the pixels in the rows/cols that are above/below the pixel to be blurred. The number of rows and columns to use in the operation is determined by a parameter called the **window size**. Each pixel in the blurred image has a value that represents the average of the pixels within the window around it from the original image:

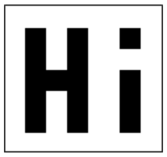


	0	1	2	3	4	5	
0	34	63	75	23	4	6	…
1	45	16	28	33	32	65	…
2	75	89	46	90	56	55	…
3	57	13	6	0	26	37	…
4	40	58	65	76	56	21	…
5	63	86	0	42	86	88	…
	⋮	⋮	⋮	⋮	⋮	⋮	⋱

	0	1	2	3	4	5	
0	?	?	?	?	?	?	…
1	?	?	?	?	40	?	…
2	?	?	?	?	?	?	…
3	?	?	49	?	?	?	…
4	?	?	?	?	?	?	…
5	?	?	?	?	?	?	…
	⋮	⋮	⋮	⋮	⋮	⋮	⋱

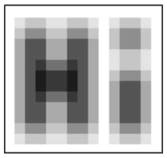
Imagine taking the following grayscale image and computing a blur for it. Here is the original image containing pixels that are either white (i.e., 255) or black (i.e., 0):

255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	0	0	255	255	0	0	255	255	255	0	0	255	255
255	255	0	0	255	255	0	0	255	255	255	0	0	255	255
255	255	0	0	255	255	0	0	255	255	255	255	255	255	255
255	255	0	0	255	255	0	0	255	255	255	255	255	255	255
255	255	0	0	0	0	0	0	255	255	255	0	0	255	255
255	255	0	0	0	0	0	0	255	255	255	0	0	255	255
255	255	0	0	255	255	0	0	255	255	255	0	0	255	255
255	255	0	0	255	255	0	0	255	255	255	0	0	255	255
255	255	0	0	255	255	0	0	255	255	255	0	0	255	255
255	255	0	0	255	255	0	0	255	255	255	0	0	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255	255



Here is the image after blurring:

255	255	255	255	255	255	255	255	255	255	255	255	255	255	255
255	227	198	198	227	227	198	198	227	255	227	198	198	227	255
255	198	142	142	198	198	142	142	198	255	198	142	142	198	255
255	170	85	85	170	170	85	85	170	255	198	142	142	198	255
255	170	85	85	170	170	85	85	170	255	227	198	198	227	255
255	170	85	57	113	113	57	85	170	255	227	198	198	227	255
255	170	85	28	57	57	28	85	170	255	198	142	142	198	255
255	170	85	28	57	57	28	85	170	255	170	85	85	170	255
255	170	85	57	113	113	57	85	170	255	170	85	85	170	255
255	170	85	85	170	170	85	85	170	255	170	85	85	170	255
255	170	85	85	170	170	85	85	170	255	170	85	85	170	255
255	198	142	142	198	198	142	142	198	255	198	142	142	198	255
255	227	198	198	227	227	198	198	227	255	227	198	198	227	255
255	255	255	255	255	255	255	255	255	255	255	255	255	255	255



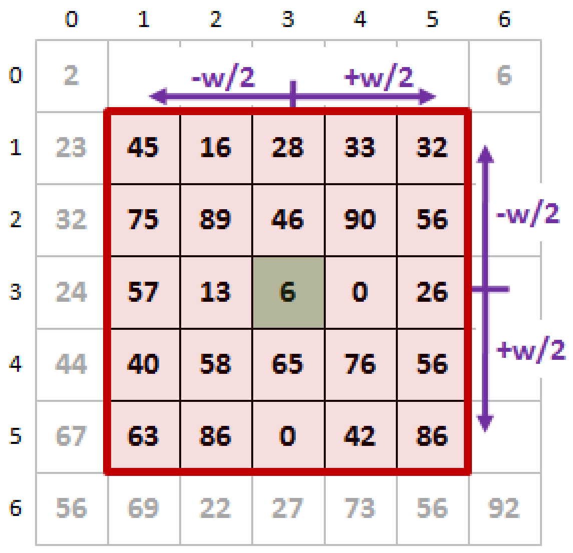
You can see that the black pixels becomes lighter due to the fact that they have incorporated the white pixels around it in its new average value. Similarly, some of the white pixels became gray due to the averaging of nearby black pixels.

How could we write the code to do this assuming that the image is grayscale and is stored in an array ? We need to go through each pixel one-by-one…so a nested **for** loop is necessary. As we move through the pixels, we need to find the average of all pixels in the window around it and then set the new image's pixel to that averaged value. Here is the basic idea, although some details still need to be worked out:

```
Algorithm: BasicBlurGrayscaleImage
        image:          the 2D array representing the image
        R:              number of rows in the image
        C:              number of columns in the image
        windowSize:     width & height of the averaging window

1.      blurredImage ← new Array with capacity image.length
2.      for each row r from 0 to R-1 {
3.              for each column c from 0 to C-1 {
4.                      sum ← 0
5.                      for each pixel p in the window around and including image[r][c] {
6.                              sum ← sum + p
                        }
7.                      blurredImage[r][c] = sum / windowSize$^2$
                }
        }
8.      image ← blurredImage
```

The loop on line 5 is not clear. How do we find each **p** pixel ?

One way is to consider the window pixels as being "offsets" in the array with respect to the pixel's location. So, for example, to get the new value for the pixel at row **3**, column **3** (see below) with a window size **w** of 5, we can consider offsets of **–w/2** to +**w/2** in both the rows and columns (which corresponds to **-2** to **+2** when **w=5**) as shown here →

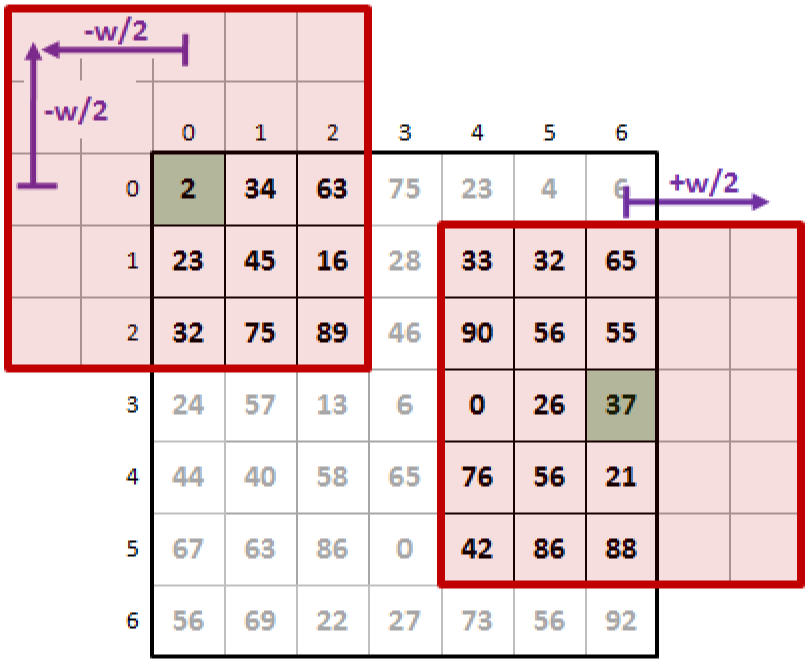


	0	1	2	3	4	5	6
0	2						6
1	23	45	16	28	33	32	
2	32	75	89	46	90	56	
3	24	57	13	6	0	26	
4	44	40	58	65	76	56	
5	67	63	86	0	42	86	
6	56	69	22	27	73	56	92

Thus, we can use a **for** loop to iterate through the **w**^{2} pixels in the window:

```
for each rowOffset from –w/2 to +w/2 {
        for each colOffset from –w/2 to +w/2 {
                … image[r+rowOffset][c+colOffset]
        }
}
```

Of course, there will be some issues along the borders of the image because adding the offsets will cause image coordinates that are less than zero or greater than the image's width or height →

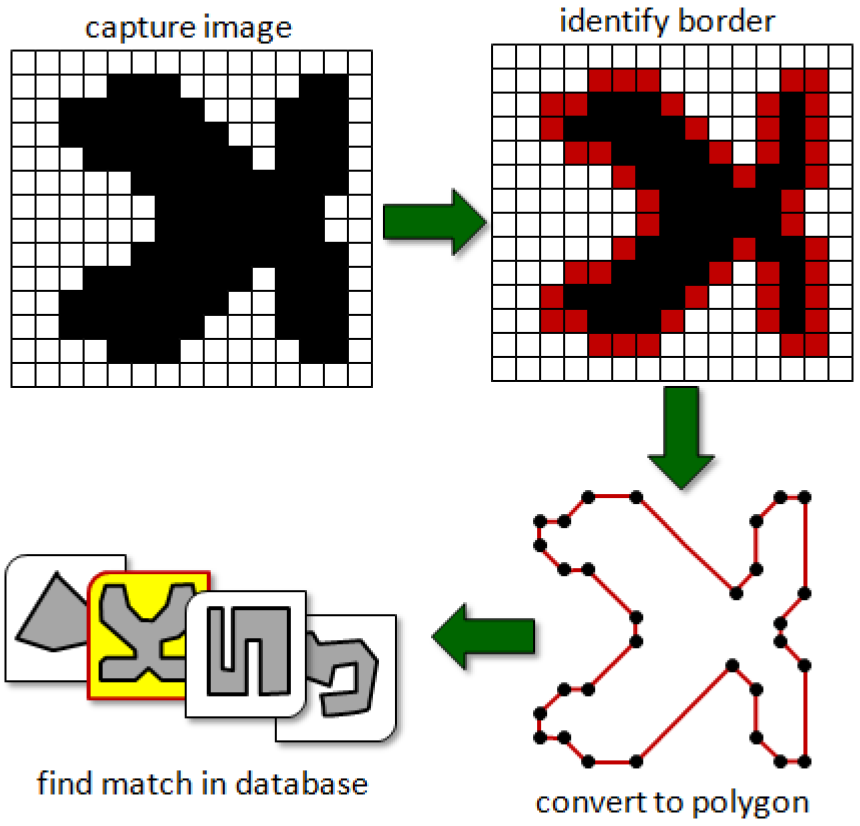


So we will have to check for these "boundary" cases. We can either ignore them (i.e., don't average any pixels that are along such a border), or we can adjust the computation so that it also counts how many pixels are within the boundary and averages just those instead. We'll take the 2nd approach. So we will need to count how many valid pixels are within the window and use that to get the average. Here is the completed solution, making sure that the nested window loops are snuggled nicely within our outer nested loops:

```
Algorithm: BlurGrayscaleImage
        image:          the 2D array representing the image
        R:              number of rows in the image
        C:              number of columns in the image
        windowSize:     width/height of the averaging window

1.      blurredImage ← new Array with capacity image.length
2.      for each row r from 0 to R-1 {
3.          for each column c from 0 to C-1 {
4.              sum ← 0
5.              count ← 0
6.              for each rowOffset from –w/2 to +w/2 {
7.                  for each colOffset from –w/2 to +w/2 {
8.                      if (((r+rowOffset) >= 0) AND (r+rowOffset) < R) AND
                           ((c+colOffset) >= 0) AND ((c+colOffset) < C)) then {
9.                          sum ← sum + image[r+rowOffset][c+colOffset]
10.                         count ← count + 1
11.                     }
12.                 }
13.             }
14.             blurredImage[r][c] = sum / count
            }
        }
15.     image ← blurredImage
```

Example:

Imagine some machine parts moving along a conveyor belt. Assume that a robot must pick up a particular type part to assemble some product. Before the parts reach the robot, a camera takes images of the moving parts. The image is processed to a black and white silhouette showing the shape of a part. We would like to process the image to find the ***edges*** of the part so that we can compare it to a database of "known" parts and identify the shape for pickup by the robot arm.

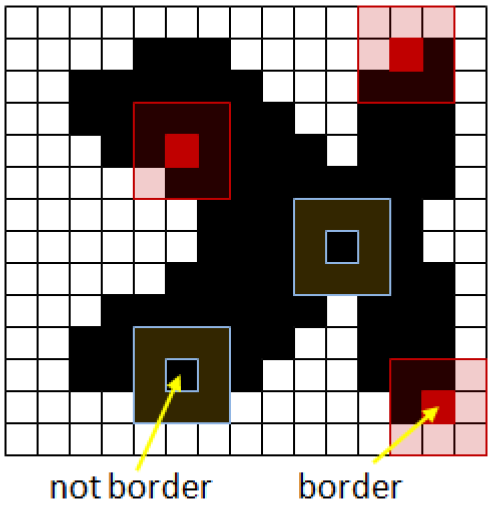


Let us consider the portion of this problem that determines the border (i.e., edge) of the object in the silhouette image.

There are many strategies for doing edge detection in an image. However, we will use an approach that computes a *chain* of pixels along an object's boundary. This will allow us to easily convert the pixel chain into a polygon and then compare with the polygons stored in the database.

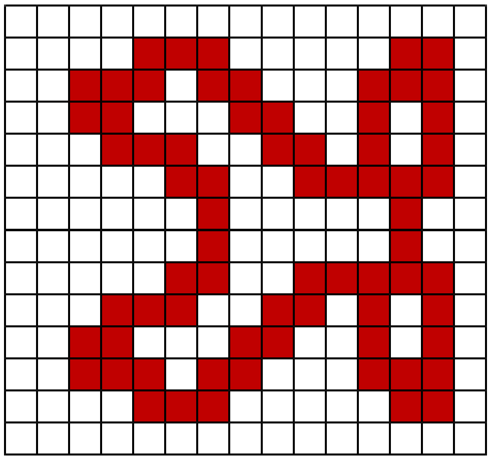
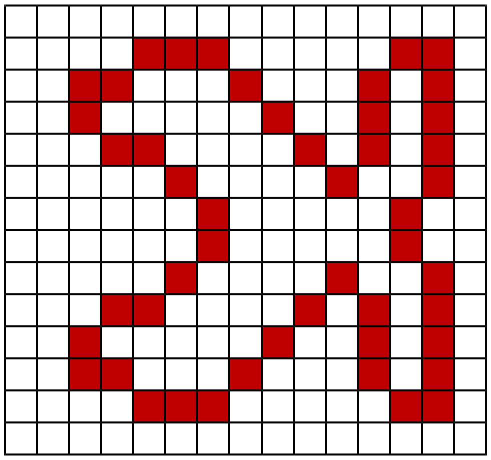
For now, assume that the image is given to us so that the entire part is within the boundaries of the image and that only one part is in the image … and the image is black and white (i.e., binary). How do we find the border pixels ?

One approach may be to simply process the image by looking at each black pixel and specifying that it is a border pixel if it is not completely surrounded by **8** black pixels all around it as shown here →



The result is that we could identify pixels that are clearly inside the part and those that are along the border.

Below, on the left, is the result. On the right is the "thinner" result that would be obtained if we "relaxed" our conditions a little and assume that a pixel is on the border only if the **4** pixels above or below are white (ignoring diagonals):

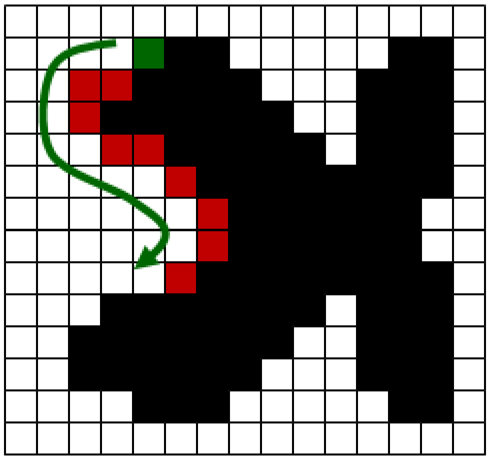
Here is the code that takes the black and white image and colors the border pixels red:

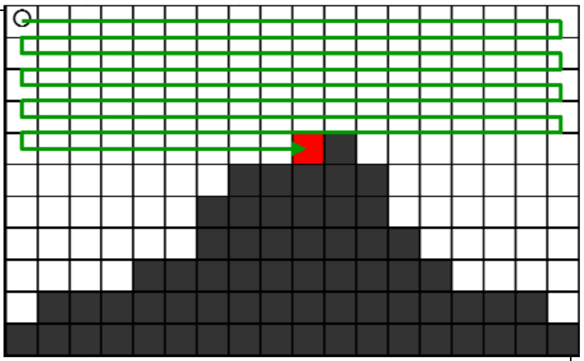
```
Algorithm: BordersGrayscaleImage
        image:          2D array representing the image
        R:              # rows in image
        C:              # columns in image

1.      blurredImage ← new Array with capacity image.length
2.      for each row r from 0 to R-1 {
3.            for each column c from 0 to C-1 {
6.                  if (image[r][c] is black) then {
6.                        if ((image[r+1][c] is white) OR (image[r-1][c] is white) OR
8.                           (image[r][c+1] is white) OR (image[r][c-1] is white)) then
9.                                image[r+1][c] ← red
13.                 }
              }
        }
```

The code above properly determines the border pixels. However, there is a problem. It does not give us the order of the pixels…so we cannot easily form a polygon (which has vertices and edges).

A better approach would be to start at some "beginning" edge pixel and then somehow trace around the border of the shape until we get back at the beginning pixel. →

What pixel do we start the tracing at ? It does not matter, but for simplicity, we can choose the top-most/left-most pixel with some code like this:

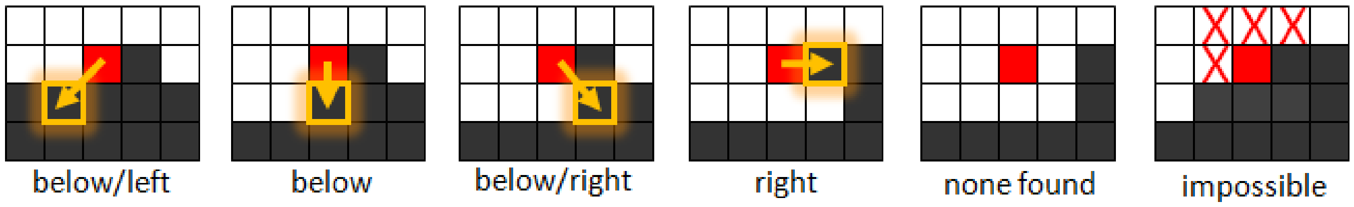




Algorithm: FindStart

```
        image:     2D array representing image
        R:         # rows in image
        C:         # columns in image

1.      for each row r from 0 to R-1 {
2.           for each column c from 0 to C-1 {
3.                if (image[r][c] is black) then {
4.                     startR← r
5.                     startC← c
                       quit with startR, startC as starting pixel
                }
           }
      }
```

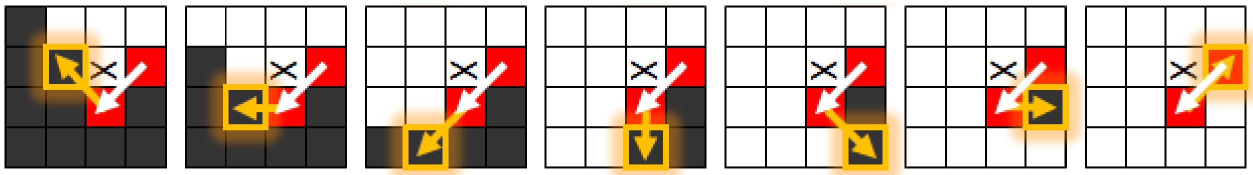
Now that we have the starting pixel, how do we start tracing ? Well, let us assume that we will trace the border counter-clockwise around the shape. Where should the next border point be ? Here are the possibilities:



The next pixel along the border depends on the shape itself. Above, we see that there are 5 possibilities … the 5th one being a very special case where the black pixel that we found was in fact a "stand-alone" black pixel … perhaps noise from the incoming data. The last image shows impossible situations … given that we started at the top left black pixel to begin with.

You may notice that all possibilities are below or to the right. You may also notice that we can simply check these 4 possibilities in sequence. Does the order matter ? Yes. We need to start by checking below/left first. Why ? Look at the last image above. What if we checked the below/right pixel first ? It is clearly not on the border of the shape.

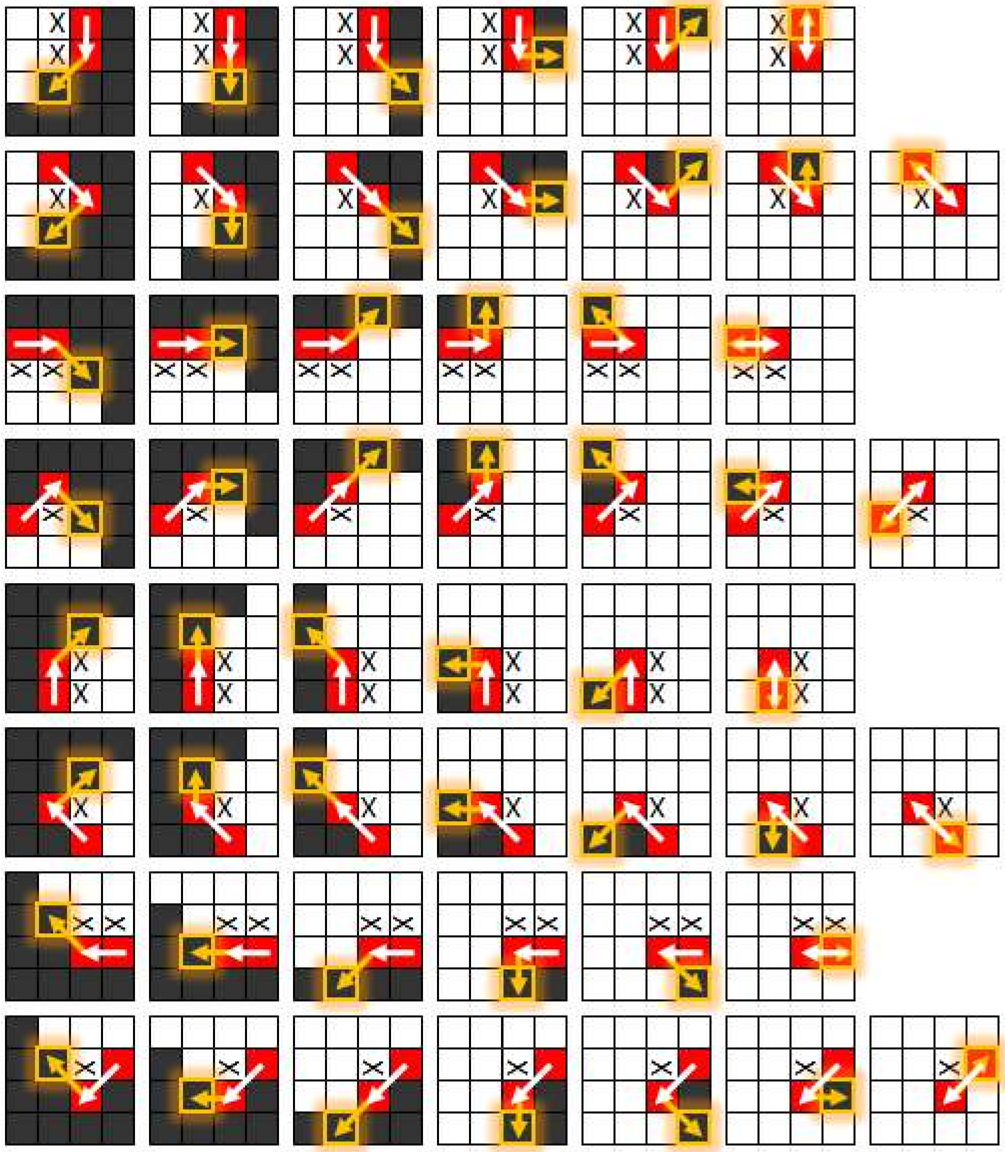
Assume then that we ended up finding a black pixel in the below/left position. We have our next point along the border. But then what ? We need to do a similar process again:



Notice now that there are more possibilities (7, in fact). So how do we know how to choose the next pixel ? We need to look around the last added pixel, again in a counter-clockwise order … this time starting with the upper-left pixel….until we find a black pixel. Notice that the pixel above (shown as X) is not a possible "next-pixel" because it would have been chosen beforehand in the border tracing.

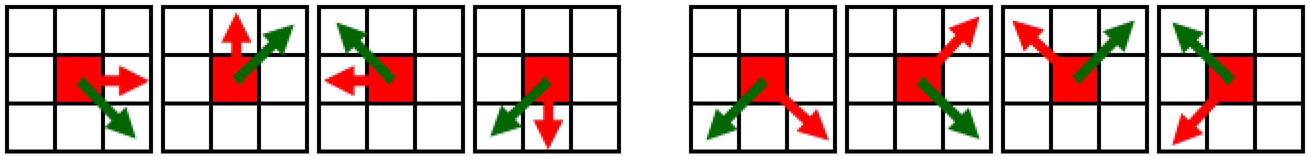
So, the first time, we started looking in the bottom left first, the next time we started in the upper left … it is not consistent. In fact, the pixel that we start looking at first (in our trace around the last added border pixel) depends on the direction that we came from.

It is not as straight-forward and intuitive as it seems. Lets draw some possibilities:



It looks like a lot of possibilities, but every second set of pictures is just a 90° rotated version of a previous set. When solving problems in computer science, it is often necessary to write out these "test cases" and to try and make sure that you cover every possible solution. In this case, there are a small number of possibilities for each situation, so this is feasible. However, in some problems, it is not possible to figure out all possible situations. In time, you will get to know which kinds of problems will have a small fixed number of possibilities and which problems are too difficult (or too time consuming) to determine all possible solutions.

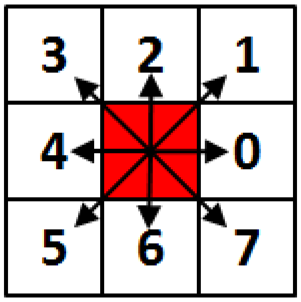
Given each of the 8 possible incoming directions, the leftmost pictures above show the direction to the next border pixel. Here is the series of images showing (red arrow) the ***incoming direction*** (from previous border pixel) and the corresponding ***start direction*** (to next potential border pixel) in which to start looking for the next border pixel:



So, to get the next pixel in the border, we could just use 8 **if** statements to figure it out:

```
if (incomingDirection is right) then
        startDirection ← lowerRight
if (incomingDirection is up) then
        startDirection ← upperRight
if (incomingDirection is left) then
        startDirection ← upperLeft
if (incomingDirection is down) then
        startDirection ← lowerLeft

if (incomingDirection is lowerRight) then
        startDirection ← lowerLeft
if (incomingDirection is upperRight) then
        startDirection ← lowerRight
if (incomingDirection is uperLeft) then
        startDirection ← upperRight
if (incomingDirection is lowerLeft) then
        startDirection ← upperLeft
```

However, there is an easier way. Consider numbering the pixels around the a pixel as shown here to the right →



In this case, the leftmost 4 pictures above will assign the start direction to be: **(*incomingDirection* + 7) modulo 8** and the rightmost pictures above will assign the start direction to be: **(*incomingDirection* + 6) modulo 8**. In both cases, the **modulo 8** ensures that the maximum value is **7**. It accounts for the "wrap around" effect.

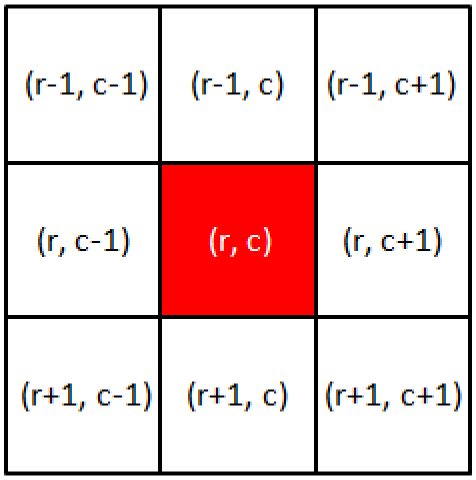
Interestingly, the 4 left pictures above all have even numbered incoming directions, while the others are odd numbered. So, we can use this code instead of the 8 **if** statements above:

```
if (incomingDirection is even) then
        startDirection ← (incomingDirection + 7) modulo 8
otherwise
        startDirection ← (incomingDirection + 6) modulo 8
```

How do we check if a number is even or odd ? We can divide by 2 and check if the remainder is 0 or 1. So, **incomingDirection modulo 2 = 0** means that the **incomingDirection** is even.

OK, so we know the direction in which to start searching for the next border pixel. How do we actually do this search ? Well, we need to search counter clockwise until we find a black pixel, or until we checked all the 6 or 7 valid positions around the previous border pixel. In fact, we can check all 8 positions…it does not really matter, but 7 is sufficient to cover both cases.

But wait. How do we even check the pixel in the **startDirection** ? Where is that pixel in the array with respect to the last border pixel ? We need a way to convert the **startDirection** into an actual row and column in the array. Assume that the last border pixel is a row **r**, column **c**. Notice here, how the offsets are set up according to position (**r**, **c**) →



(r-1, c-1)	(r-1, c)	(r-1, c+1)
(r, c-1)	(r, c)	(r, c+1)
(r+1, c-1)	(r+1, c)	(r+1, c+1)

Again, we could set up a set of 8 if statements

```
if (startDirection is 0) then {
        startR ← r
        startC ← c + 1
}
if (startDirection is 1) then {
… etc…
```

However, there is a shorter way. We can set up 2 arrays of constant offsets and then access them using the **startDirection** as an index. This is called a ***lookup table***.

Here are the tables:

```
rowOffsets = new array with numbers: [0, -1, -1, -1, 0, 1, 1, 1]
colOffsets = new array with numbers:  [1, 1, 0, -1, -1, -1, 0, 1]
```

Then we simply compute the next potential border pixel as:

```
image[r + rowOffsets[startDirection]][c + colOffsets[startDirection]]
```

Now all we need to do is to loop counter-clockwise around the last border pixel, looking for a black pixel, starting with the **incomingDirection**. Here is a function that takes the last-added pixel (**r**, **c**) in the image along the border that is being traced and a starting **incomingDirection** and then determines the next pixel in the image.

The code returns a new point which represents the location of the next pixel along the border during the tracing procedure. It also updates the **incomingDirection** to be the last **startingDirection** in order to get ready for the next pixel along the border trace.

```
Algorithm: FindNextPixel
      image:                  2D array representing the image
      incomingDirection:      direction coming in to this pixel
      r:                      row# of pixel in image
      c:                      column# of pixel in image

1.    rOff = new array with numbers: [0, -1, -1, -1, 0, 1, 1, 1]
2.    cOff = new array with numbers:  [1, 1, 0, -1, -1, -1, 0, 1]
3.    if (incomingDirection modulo 2 is 0) then
4.          startDirection ← (incomingDirection + 7) modulo 8
      otherwise
5.          startDirection ← (incomingDirection + 6) modulo 8

6.    count ← 0
7.    found ← false
8.    while (count < 8 AND found is false) {
9.          if (image[r + rOff[startDirection]][c + cOff[startDirection]] is black) then
10.               found ← true
            otherwise {
11.               count ← count + 1
12.               startDirection ← (startDirection + 1) modulo 8
            }
      }
13.   incomingDirection ← startDirection
14.   if (found is false) then
15.         return null
      otherwise
16.         return new point(r+rOff[startDirection], c+cOff[startDirection])
```

Now we have everything that we need for our algorithm. We know how to find the starting pixel and then to determine the direction to get the next pixel. We can do this until the start pixel is found again. We would need to set the **incomingDirection** to 7 as a start, since we started looking for the first black pixel from the top/left in the image. Here is the completed code:

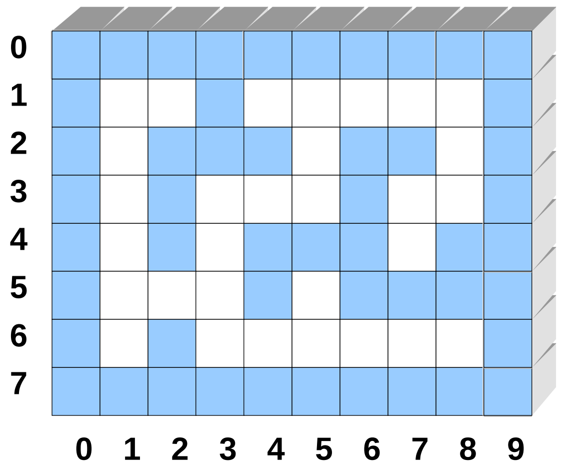
```
Algorithm: TraceBorder
        image:          2D array representing the image
        R:              # rows in image
        C:              # columns in image

1.      polygon ← new polygon with no points
2.      startPixel ← null
3.      for each row r from 0 to R-1 {
4.          for each column c from 0 to C-1 {
5.              if ((startPixel is null) AND (image[r][c] is black)) then {
6.                  startPixel ← (r, c)
7.                  add startPixel to polygon
                }
            }
        }
8.      if (startPixel is not null)) then {      // handles case where image is all white
9.          direction ← 7
10.         done ← false
11.         while (done is false) {
12.             nextPixel ← findNextPixel(image, direction, r, c)
13.             if ((nextPixel is null) OR (nextPixel is same as startPixel)) then
14.                 done ← true
                otherwise
15.                 add nextPixel to polygon
            }
        }
16.     return polygon
```

Notice that lines 2 through 7 simply find the starting pixel (if there is one), while lines 9 through 15 do the tracing along the border. Each time, a new border pixel (i.e., **nextPixel**) is added to the polygon until the original **startPixel** is reached again. The code will also handle the case where the **startPixel** is the only black pixel…in this case **nextPixel** will be **null**.

Example:

The examples that we have seen so far are related to image processing. However, 2D arrays can be used for any situation in which data is arranged in a grid. For example, consider representing the following maze by using arrays →

In Processing, we could represent this maze by using a 2D array of **ints** indicating whether or not there is a wall at each location in the array (i.e., 1 for wall, 0 for open space).



Notice how we can do this by using the quick array declaration with the braces **{ }** as we did with one-dimensional (i.e., 1D) arrays:

```
     int[][]      maze = {{1,1,1,1,1,1,1,1,1,1},
                          {1,0,0,1,0,0,0,0,0,1},
                          {1,0,1,1,1,0,1,1,0,1},
                          {1,0,1,0,0,0,1,0,0,1},
                          {1,0,1,0,1,1,1,0,1,1},
                          {1,0,0,0,1,0,1,1,1,1},
                          {1,0,1,0,0,0,0,0,0,1},
                          {1,1,1,1,1,1,1,1,1,1}};
```

This array represents a grid with 8 rows and 10 columns. Notice that there are more brace characters than with 1D arrays. Each row is specified by its own unique braces and each row is separated by a comma. In fact, each row is itself a 1D array.

We could display this array quite simply by iterating through the rows and columns:

```
for (int row=0; row<8; row++) {
    for (int col=0; col<10; col++)
       print(maze[row][col]);
    println();
}
```

```
1111111111
1001000001
1011101101
1010001001
1010111011
1000101111
1010000001
1111111111
```

Of course, we may want to display the maze with nicer-looking characters:

```
for (int row=0; row<8; row++) {
    for (int col=0; col<10; col++) {
        if (item == 1)
          print('*');
        else
          print(' ');
    }
    println();
}
```

```
**********
*  *     *
* *** ** *
* *   *  *
* * *** **
*   * ****
* *      *
**********
```

How though, would we draw the maze (as shown in the picture above) on the window in processing ? We would need to decide upon how big each square would be and then loop through drawing the squares in the appropriate color.

Here is a more complete program (assuming that the **maze** variable is set as shown earlier):

```
int  GRID_SIZE = 20;

void setup() {
  size(GRID_SIZE*10,GRID_SIZE*8);
  stroke(0);   // use a black border
}

void draw() {
  drawMaze();
}
```

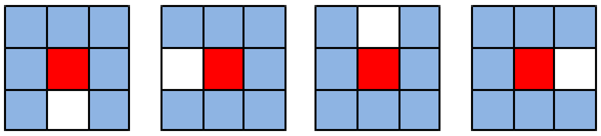
```
void drawMaze() {
  for (int row=0; row<8; row++) {
    for (int col=0; col<10; col++) {
        if (maze[row][col] == 1)
           fill(150,200,255);    // light blue
        else
          fill(255, 255, 255);   // white
        rect(col*GRID_SIZE, row*GRID_SIZE, GRID_SIZE, GRID_SIZE);
    }
  }
}
```

Notice how the **GRID_SIZE** is an adjustable parameter which is used as a scalar to make the maze appear larger or smaller.

Example:

Some mazes contain "dead-ends". A "dead-end" is any location in the maze that has only one way to get into it (i.e., it is surrounded by 3 walls). Using the previous maze example, how can we adjust the code so that dead-ends are colored red ?

We need to identify a dead-end by checking the grid locations around it. There are exactly 4 cases that we should check as shown here →



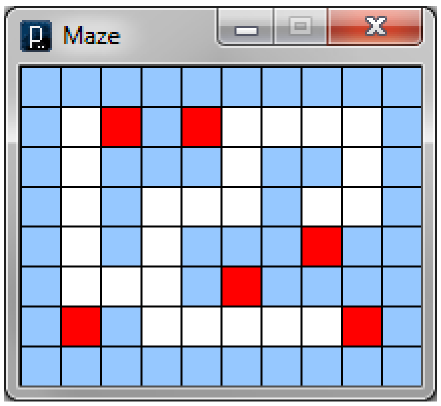
Given a particular grid location **maze[r][c]** that is an open space (i.e., **maze[r][c] = 0**) we can determine if it is a dead-end like this:

```
if (((maze[r-1][c] is a wall) AND (maze[r][c-1] is a wall) AND (maze[r][c+1] is a wall)) OR
   ((maze[r-1][c] is a wall) AND (maze[r+1][c] is a wall) AND (maze[r][c+1] is a wall)) OR
   ((maze[r+1][c] is a wall) AND (maze[r][c-1] is a wall) AND (maze[r][c+1] is a wall)) OR
   ((maze[r-1][c] is a wall) AND (maze[r+1][c] is a wall) AND (maze[r][c-1] is a wall))) then
       maze[r][c] is a dead end, so paint it red
```

This code checks each of the 4 cases by checking the grid locations above, below, to the right and to the left of location (r,c). Can we do this in a simpler way, given that all maze locations are represented as numbers either 0 or 1 ?

Yes. We can add up the maze locations of the 4 directions around (r,c). If their sum is 3, then this is a dead-end.

```
count ← maze[r-1][c] + maze[r+1][c] + maze[r][c-1] + maze[r][c+1]
if (count is 3) then
    maze[r][c] is a dead end, so paint it red
```



Of course, this assumes that **(r,c)** is not along the border, otherwise we may be trying to examine locations that are outside of the maze boundaries (e.g., r-1<0, c-1<0, r+1>7, c+1>7).

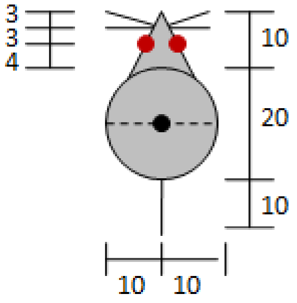
Of course, in our maze example, there are no open (i.e., white) locations along the border, so we would not need to worry about going beyond the boundaries. If however, the maze had openings along the border, we would have to check for such boundary issues like this:

```
if ((r>0) AND (r<7) AND (c>0) AND (c<9)) then {
        count ← maze[r-1][c] + maze[r+1][c] +
                    maze[r][c-1] + maze[r][c+1]
        if (count is 3) then
                maze[r][c] is a dead end, so paint it red
}
```

Example:

Assume that we had an underlying maze as in our previous example and that we wanted to move an object (e.g., a small rat) around in the maze by clicking on successive adjacent maze locations to move the rat.

To begin, here is what our rat will look like →



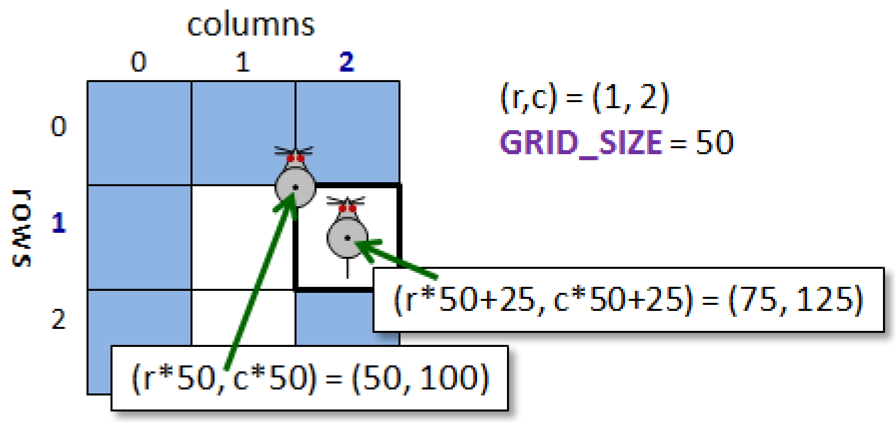
As we learned earlier in the course, the code for drawing this rat at a particular location is as easy as drawing a combination of circles, a triangle and some lines.

However, we would like to keep track of the rat's location in terms of the row and column that it lies in within the maze. Therefore, we will assume that the rat's position is recorded as a **(row,col)** in the maze and that we need to calculate the **(x,y)** position based on this row and column.

The **x** and **y** position of the rat's center (shown as the black dot above) can be computed as:

```
x ← col * GRID_SIZE  + (GRID_SIZE /2)
y ← row * GRID_SIZE + (GRID_SIZE /2)
```

The **GRID_SIZE** is the width and height (in pixels) of each square grid location. So, multiplying the row and column by the **GRID_SIZE** gives us the **(x,y)** location of the top left corner of the grid location. However, we likely want to draw the rat in the center of the grid location, so we need to add half of the **GRID_SIZE**:



Here is the code, then, for drawing the rat:

```
int  ratRow, ratCol;

void drawRat() {
  int   x = ratCol * GRID_SIZE + GRID_SIZE/2;
  int   y = ratRow * GRID_SIZE + GRID_SIZE/2;
  fill(150, 150, 150);  // gray
  triangle(x, y-20, x+10, y, x-10, y);      // head
  line(x, y+10, x, y+20);                   // tail
  line(x, y-17, x-10, y-17);                // whisker1
  line(x, y-17, x-10, y-20);                // whisker2
  line(x, y-17, x+10, y-17);                // whisker3
  line(x, y-17, x+10, y-20);                // whisker4
  ellipse(x, y, 20, 20);                    // body
  fill(255, 0, 0);  // red
  ellipse(x-3, y-14, 3, 3);                 // eye1
  ellipse(x+3, y-14, 3, 3);                 // eye2
}
```

How can we write the program that draws the rat at the location clicked on ?

Well, we know how to draw the maze and the rat… we just need to handle mouse-press events. At any time, the **draw()** method should always draw the rat at its current location in the grid. We only need to update that location when the user clicks on a new location.

```
int  GRID_SIZE = 50;
int  ratRow = 1, ratCol = 1;

void setup() { ... }
void drawMaze() { ... }
void drawRat() { ... }

void draw() {
  drawMaze();
  drawRat();
}

void mousePressed() {
      ... write code here ...
}
```

So, what do we do when the mouse is pressed ? We simply determine the **ratRow** and **ratCol** that the user clicked on. Once we change these variables to the new location, the **draw()** procedure should automatically draw the rat at the new location.

Determining the **(row, column)** from the **(mouseX, mouseY)** is not difficult. Recall that we found the **(x,y)** to draw the rat by multiplying the columns and rows by the **GRID_SIZE**. To do the reverse, we simply divide by the **GRID_SIZE**:

```
void mousePressed() {
  ratRow = mouseY / GRID_SIZE;
  ratCol = mouseX / GRID_SIZE;
}
```

Of course, we need to make sure that we move the rat to an "open" grid location. The rat should be allowed to move to any grid location that is not a wall:

```
void mousePressed() {
  int newRow = mouseY/GRID_SIZE;
  int newCol = mouseX/GRID_SIZE;
  if (maze[newRow][newCol] == 0) {
     ratRow = newRow;
     ratCol = newCol;
  }
}
```

If we are not simply selecting a location for the rat, but in fact steering it around the maze, we will want to ensure that we only move the rat to an adjacent grid location (i.e., up, down, left or right). Given that the rat is at location (r,c), the valid new locations are (r-1, c), (r+1, c), (r, c-1) and (r, c+1). Of course (r,c) is also a valid location, but the rat does not need to change locations in that case. Here is the code:

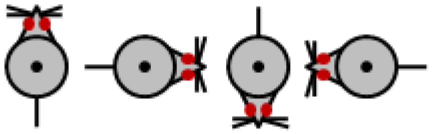
```
void mousePressed() {
  int newRow = mouseY/GRID_SIZE;
  int newCol = mouseX/GRID_SIZE;
  if (maze[newRow][newCol] == 0) {
     if (((newRow == ratRow+1) && (newCol == ratCol)) ||
         ((newRow == ratRow-1) && (newCol == ratCol)) ||
         ((newRow == ratRow) && (newCol == ratCol+1)) ||
         ((newRow == ratRow) && (newCol == ratCol-1))) {
        ratRow = newRow;
        ratCol = newCol;
     }
  }
}
```

Can this code be reduced ? Yes. We can make use of the **abs** function by examining the difference between the old row and the new row (same for columns). If the difference between the old and new row values is 1 and the column difference is 0, then this is a valid new position. Likewise, if the difference in columns is 1 and the rows is zero, then this too is a good position. So, we can add up the differences in rows and columns and make sure that the value is **1**.

```
void mousePressed() {
  int newRow = mouseY/GRID_SIZE;
  int newCol = mouseX/GRID_SIZE;
  if (maze[newRow][newCol] == 0) {
     if (abs(newRow-ratRow) + abs(newCol-ratCol) == 1) {
        ratRow = newRow;
        ratCol = newCol;
     }
  }
}
```

Example:

How can we adjust this program so that the rat turns to face the direction that it just travelled ?



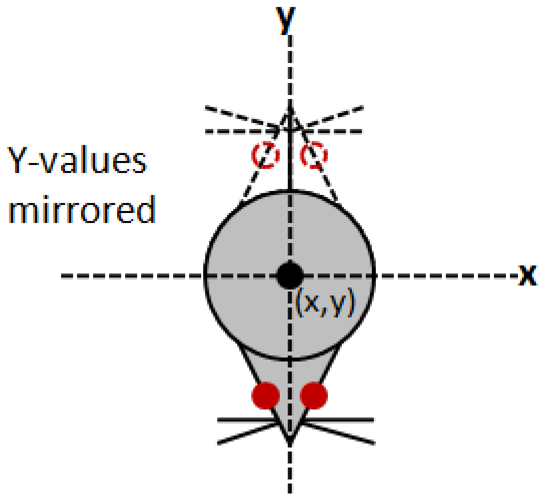
Since the rat travels in just 4 directions, we need to display it in one of 4 possible positions as shown here.

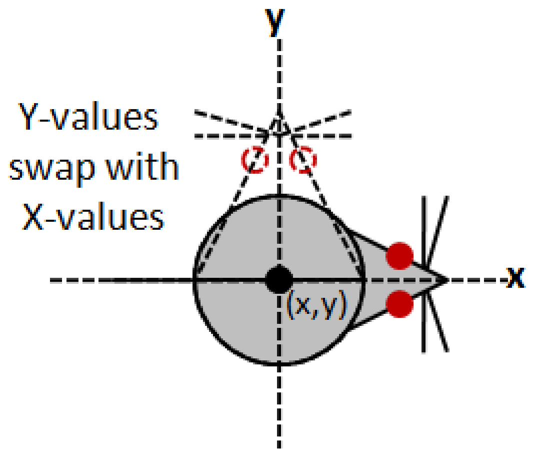
The only difference in the above code will be with respect to the **drawRat()** procedure. Recall that this code draws the rat centered around the **x** and **y** location of the center of a grid square, which is computed from the **ratRow** and **ratCol** variables:

Here is the code for drawing the rat facing upwards:

```
void drawRat() {
  int   x = ratCol * GRID_SIZE + GRID_SIZE/2;
  int   y = ratRow * GRID_SIZE + GRID_SIZE/2;
  fill(150, 150, 150);  // gray
  triangle(x, y-20, x+10, y, x-10, y);       // head
  line(x, y+10, x, y+20);                    // tail
  line(x, y-17, x-10, y-17);                 // whisker1
  line(x, y-17, x-10, y-20);                 // whisker2
  line(x, y-17, x+10, y-17);                 // whisker3
  line(x, y-17, x+10, y-20);                 // whisker4
  ellipse(x, y, 20, 20);                     // body
  fill(255, 0, 0);  // red
  ellipse(x-3, y-14, 3, 3);                  // eye1
  ellipse(x+3, y-14, 3, 3);                  // eye2
}
```

What changes when we want the rat to face downwards ? In reality, it is only the **y** offset values that need to be negated (i.e., mirrored with respect to the (x,y) origin). However, when we want to face the mouse right or left, the X and Y values of the various points will need to be swapped:



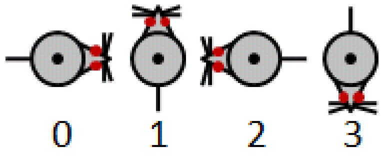


So, for example, looking at the code for drawing the triangle, here is how it changes depending on the direction that we want to draw the rat:

```
triangle(x+20, y, x, y-10, x, y+10);        // facing right
triangle(x, y-20, x+10, y, x-10, y);        // facing upwards
triangle(x-20, y, x, y+10, x, y-10);        // facing left
triangle(x, y+20, x-10, y, x+10, y);        // facing downwards
```

Since there are many coordinates that need to change like this (i.e., the line endpoints and ellipse centers), we will need to change the constants throughout the **drawRat()** procedure depending on the direction of the rat.

The simplest way to write the code is to duplicate the code 4 times (i.e., once for each direction). This, however, is a lot of duplication. We can do better. We can number the directions as shown here →



We would thus need to create the following variable to store the rat's current direction:

```
int  direction;    // 0 = right, 1 = up, 2 = left, 3 = down
```

Then, a better way to write the code would be to take advantage of the fact that directions 0 and 2 have negated offsets as well as directions 1 and 3. So, we can combine two of these by creating a multiplication factor **f** with a value of either 1 or -1 as follows:

```
void drawRat() {
  int   f = 1;
  int   x = ratCol*GRID_SIZE + (GRID_SIZE /2);
  int   y = ratRow*GRID_SIZE + (GRID_SIZE /2);
  fill(150, 150, 150);   // gray

  if ((direction == 1) || (direction == 2))
    f = -1;

  if ((direction == 1) || (direction == 3)) {
    triangle(x, y+f*20, x+10, y, x-10, y);
    line(x, y-f*10, x, y-f*20);     // tail
    line(x, y+f*17, x-10, y+f*17);  // whisker1
```

```
        line(x, y+f*17, x-10, y+f*20);  // whisker2
        line(x, y+f*17, x+10, y+f*17);  // whisker3
        line(x, y+f*17, x+10, y+f*20);  // whisker4
        ellipse(x, y, 20, 20);
        fill(255, 0, 0);   // red
        ellipse(x-3, y+f*14, 3, 3);
        ellipse(x+3, y+f*14, 3, 3);
      }
      else {
        triangle(x+f*20, y, x, y+10, x, y-10);
        line(x-f*10, y, x-f*20, y);  // tail
        line(x+f*17, y, x+f*17, y-10);  // whisker1
        line(x+f*17, y, x+f*20, y-10);  // whisker2
        line(x+f*17, y, x+f*17, y+10);  // whisker3
        line(x+f*17, y, x+f*20, y+10);  // whisker4
        ellipse(x, y, 20, 20);
        fill(255, 0, 0);   // red
        ellipse(x+f*14, y-3, 3, 3);
        ellipse(x+f*14, y+3, 3, 3);
      }
    }
```

This is not too bad as it requires half the code than if we were to duplicate the drawing code 4 times. Perhaps, an even better way may be to create a 2D array of these constant offsets where each row of constants refers to a rat direction. Here are two such arrays representing the **x** and **y** offsets, respectively:

```
int[][] xOff = {{20, 0, 0, -10, -20, 17, 17, 17, 20, 17, 17, 17, 20, 14, 14},
                {0, 10, -10, 0, 0, 0, -10, 0, -10, 0, 10, 0, 10, -3, 3},
                {-20, 0, 0, 10, 20, -17, -17, -17, -20, -17, -17, -17, -20, -14, -14},
                {0, -10, 10, 0, 0, 0, 10, 0, 10, 0, -10, 0, -10, 3, -3}};

int[][] yOff = {{0, 10, -10, 0, 0, 0, -10, 0, -10, 0, 10, 0, 10, -3, 3},
                {-20, 0, 0, 10, 20, -17, -17, -17, -20, -17, -17, -17, -20, -14, -14},
                {0, -10, 10, 0, 0, 0, 10, 0, 10, 0, -10, 0, -10, 3, -3},
                {20, 0, 0, -10, -20, 17, 17, 17, 20, 17, 17, 17, 20, 14, 14}};
```

Now we can easily make use of these offsets within a simpler procedure as follows:

```
void drawRat() {
  int   x = ratCol * GRID_SIZE + GRID_SIZE/2;
  int   y = ratRow * GRID_SIZE + GRID_SIZE/2;
  fill(150, 150, 150);  // gray
  int[]  xs = xOff[direction];
  int[]  ys = yOff[direction];
  triangle(x+xs[0], y+ys[0], x+xs[1], y+ys[1], x+xs[2], y+ys[2]);
  line(x+xs[3], y+ys[3], x+xs[4], y+ys[4]);     // tail
  line(x+xs[5], y+ys[5], x+xs[6], y+ys[6]);     // whisker1
  line(x+xs[7], y+ys[7], x+xs[8], y+ys[8]);     // whisker2
  line(x+xs[9], y+ys[9], x+xs[10], y+ys[10]);   // whisker3
  line(x+xs[11], y+ys[11], x+xs[12], y+ys[12]); // whisker4
  ellipse(x, y, 20, 20);
  fill(255, 0, 0);   // red
  ellipse(x+xs[13], y+ys[13], 3, 3);
  ellipse(x+xs[14], y+ys[14], 3, 3);
}
```

As you can see, now no code is duplicated, but it does require some extra offset numbers.

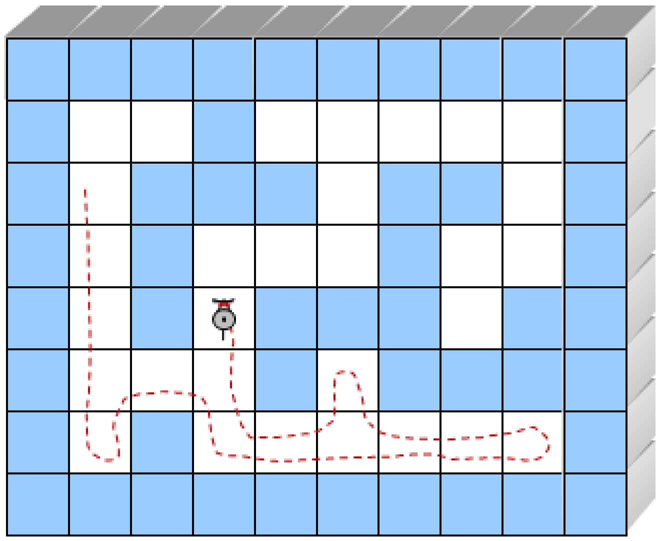
Interestingly, if we have a fixed number of directions, we can simply determine the **x** and **y** offsets for each additional direction and then increase the number of rows in the **xOff** and **yOff** arrays. Alternatively, we can calculate the **x** and **y** offsets for an arbitrary direction by using trigonometry. Then, we would not need the offset arrays. However, our code would become more complex and slower, requiring trigonometric calculations for each point of the rat.

All that remains to do is to adjust the **direction** as the user clicks on the rat's new location. Recall the code for handling mouse presses. When we find the new location to move to, we can simply compare the row and column that we are arriving at with the row and column we are leaving and this will determine the direction:

```
void mousePressed() {
  int newRow = mouseY/GRID_SIZE;
  int newCol = mouseX/GRID_SIZE;
  if (maze[newRow][newCol] == 0) {
     if (abs(newRow-ratRow) + abs(newCol-ratCol) == 1) {
        if (newRow > ratRow) direction = 3;
        else if (newRow < ratRow) direction = 1;
        else if (newCol > ratCol) direction = 0;
        else if (newCol < ratCol) direction = 2;
        ratRow = newRow;
        ratCol = newCol;
     }
  }
}
```

Example:

As a final example, how can we get the rat to travel through the maze on its own ? Assume that we simply want the rat to continuously travel around the maze without stopping. How can we do this ?

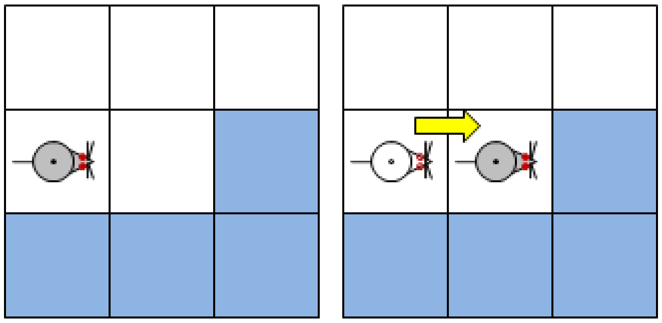


One well-known method of traveling through a maze is that of using the “right-hand rule”. The right hand rule says that as long as we keep our right hand touching a wall as we walk, we will find a solution through the maze.

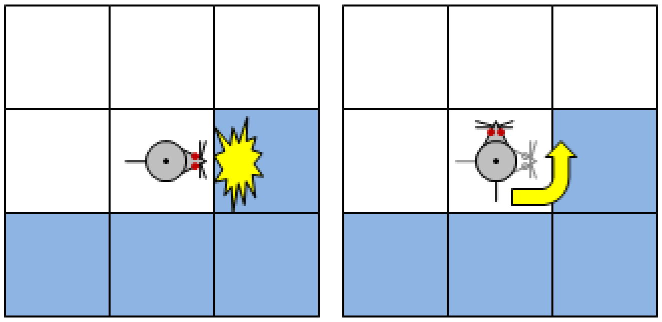
Does this work for all mazes ? I guess it depends on what we call a “maze”.

Mazes with inner loops cannot necessarily be solved using this strategy … it would depend on where the rat began following the walls.

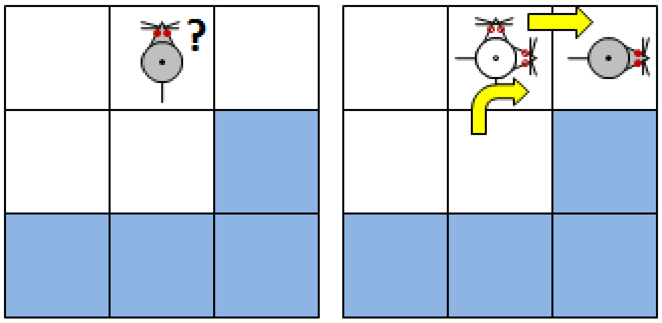
Let us consider how the rat will move around in the maze to follow the “right-hand rule”. Assume that the rat has its “right hand” on a wall. If it is able to move forward, it should simply do so as shown here →



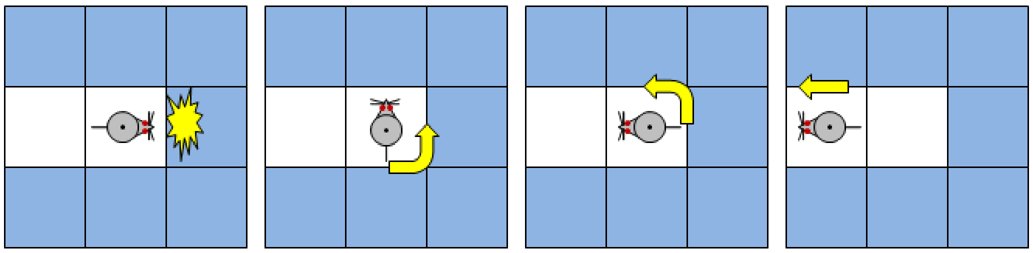
If however, the rat encounters a wall in front of it, then it should simply turn left →



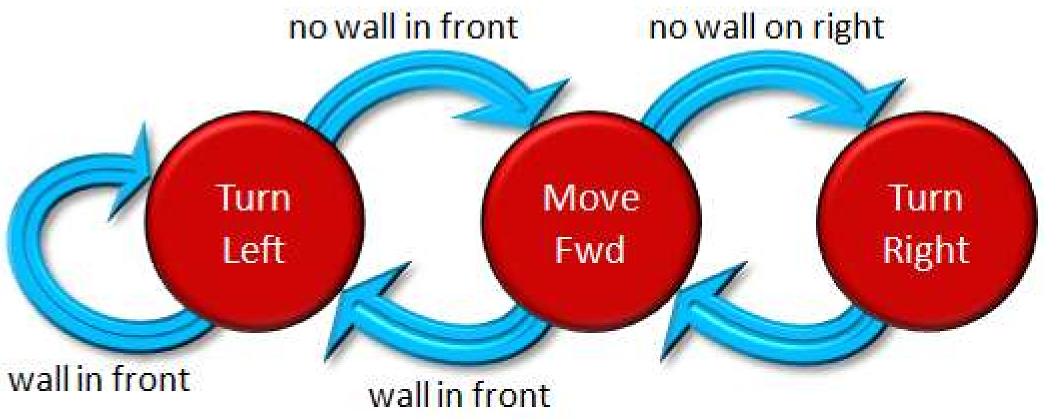
Then the rat will continue moving forwards as it now has its right hand on the wall again. It is possible that while traveling along it may lose contact with the wall →



In this case, notice that the rat needs to regain contact again with the wall on its right. First, it must turn right. However, the wall will still not be on the right of the rat after turning. Therefore it also needs to move forward. Then, it will be back-on-track again. These three cases actually encompass all situations. For example, if the rat ends up in a “dead-end”, these three cases will get it out again.



So, how can we write an algorithm for traveling through this maze ? We simply follow the wall-following model that we developed and produce a state machine diagram:



The code follows directly from the state diagram:

Algorithm: TravelMaze

```
1.      repeat {
2.          if (there is a wall on the right) then {
3.              if (there is a wall in front) then
4.                  turnLeft()
5.              otherwise
6.                  moveForward()
            }
7.          otherwise {
8.              turnRight()
9.              moveForward()
10.         }
        }
```

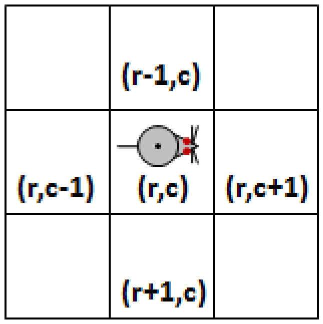
The code is straight forward. However, some details have been left out. For example, how do we know if there is a wall on the right or not ? It depends on the rat's current location in the maze as well as its current direction.

Assume that the rat has the direction and location defined in variables as before:

```
    int  ratRow, ratCol, direction;
```

Also, assume that the maze has either 1 or 0 at each (row, column) location representing walls or open spaces, respectively.

How do we determine whether or not there is a "wall on the right" of the rat ? We would need to look at the position to its right in the maze. Assuming that the rat is at position **(r, c)**, then the relative positions around it are shown here →



So, we could write code like this:

```
if (((direction is 0) AND (maze[ratRow+1][ratCol] is 1)) OR
    ((direction is 1) AND (maze[ratRow][ratCol+1] is 1)) OR
    ((direction is 2) AND (maze[ratRow-1][ratCol] is 1)) OR
    ((direction is 3) AND (maze[ratRow][ratCol-1] is 1))) then {
```

To check whether or not there is a wall to the right of the rat's location. A similar check can be made for walls in front of the rat.

How do we make a turn ? We simply increase or decrease the direction. We can do this as follows:

```
    direction ← (direction + 1) modulo 4;          // to turn left
    direction ← (direction + 3) modulo 4;          // to turn right
```

As with checking maze locations right or ahead, moving forward will depend on the direction of the rat as well:

```
    if (direction is 0) ratCol ← ratCol + 1
    else if (direction is 1) ratRow ← ratRow - 1
    else if (direction is 2) ratCol ← ratCol - 1
    else if (direction is 3) ratRow ← ratRow + 1
```

So, we can merge all of this code together to come up with a complete solution. The solution here has duplicated code. It is possible to reduce this code quite easily by making use of procedures. See if you can shrink the code.

```
Algorithm: TravelMaze
        maze:                   the maze to travel through

1.      ratRow, ratCol ← any valid starting location in the maze with wall on right
2.      direction ← any valid starting direction in the maze with wall on right

3.      repeat {
4.          if (((direction is 0) AND (maze[ratRow+1][ratCol] is 1)) OR
5.             ((direction is 1) AND (maze[ratRow][ratCol+1] is 1)) OR
6.             ((direction is 2) AND (maze[ratRow-1][ratCol] is 1)) OR
7.             ((direction is 3) AND (maze[ratRow][ratCol-1] is 1))) then {

8.              if (((direction is 0) AND (maze[ratRow][ratCol+1] is 1)) OR
9.                 ((direction is 1) AND (maze[ratRow-1][ratCol] is 1)) OR
10.                ((direction is 2) AND (maze[ratRow][ratCol-1] is 1)) OR
11.                ((direction is 3) AND (maze[ratRow+1][ratCol] is 1))) then

12.                 direction ← (direction + 1) modulo 4;        // turn left

                otherwise {
13.                 if (direction is 0) then
14.                     ratCol ← ratCol + 1
15.                 otherwise if (direction is 1) then
16.                     ratRow ← ratRow – 1
17.                 otherwise if (direction is 2) then
18.                     ratCol ← ratCol – 1
19.                 otherwise if (direction is 3) then
20.                     ratRow ← ratRow + 1
                }
            }
            otherwise {
21.             direction ← (direction + 3) modulo 4;        // turn right
22.             if (direction is 0) then
23.                 ratCol ← ratCol + 1
24.             otherwise if (direction is 1) then
25.                 ratRow ← ratRow – 1
26.             otherwise if (direction is 2) then
27.                 ratCol ← ratCol – 1
28.             otherwise if (direction is 3) then
29.                 ratRow ← ratRow + 1
            }
        }
```

Chapter 6

Sorting

What is in This Chapter ?

Sorting is a fundamental problem-solving "tool" in computer science which can greatly affect an algorithm's efficiency. Sorting is discussed in this chapter as it pertains to the area of computer science. A few sorting strategies are discussed (i.e., bubble sort, selection sort, insertion sort and counting sort) and briefly compared. Finally, sorting is applied to the problem of simulating a fire spreading across a forest.

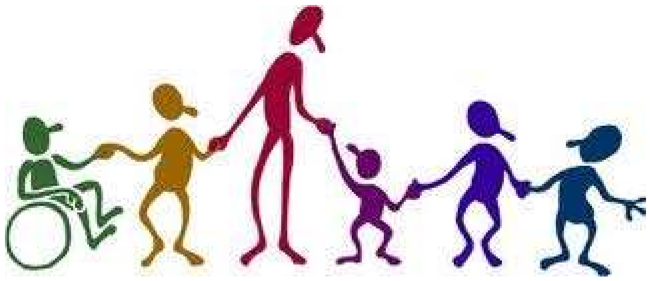
6.1 Sorting

In addition to searching lists, *sorting* is one of the most fundamental "tools" that a programmer can use to solve problems.

***Sorting** is the process of arranging items in some sequence and/or in different sets.*

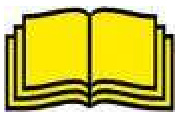
In computer science, we are often presented with a list of data that needs to be sorted. For example, we may wish to sort a list of people. Naturally, we may imagine a list of people's names sorted by their last names. This is very common and is called a ***lexicographical*** (or *alphabetical*) sorting. The "way" in which we compare any two items for sorting is defined by the ***sort order***. There are many other "sort orders" to sort a list of people. Depending on the application, we would choose the most applicable sorting order:

- sort by **ID numbers**
- sort by **age**
- sort by **height**
- sort by **weight**
- sort by **birth date**
- etc..



A list of items is just one obvious example of where sorting is often used. However, there are many problems in computer science in which it is less obvious that sorting is required. However, sorting can be a necessary first step towards solving some problems efficiently. Once a set of items is sorted, problems usually become easier to solve. For example,

- **Phone books** are sorted by name so it makes it easier to find someone's number.



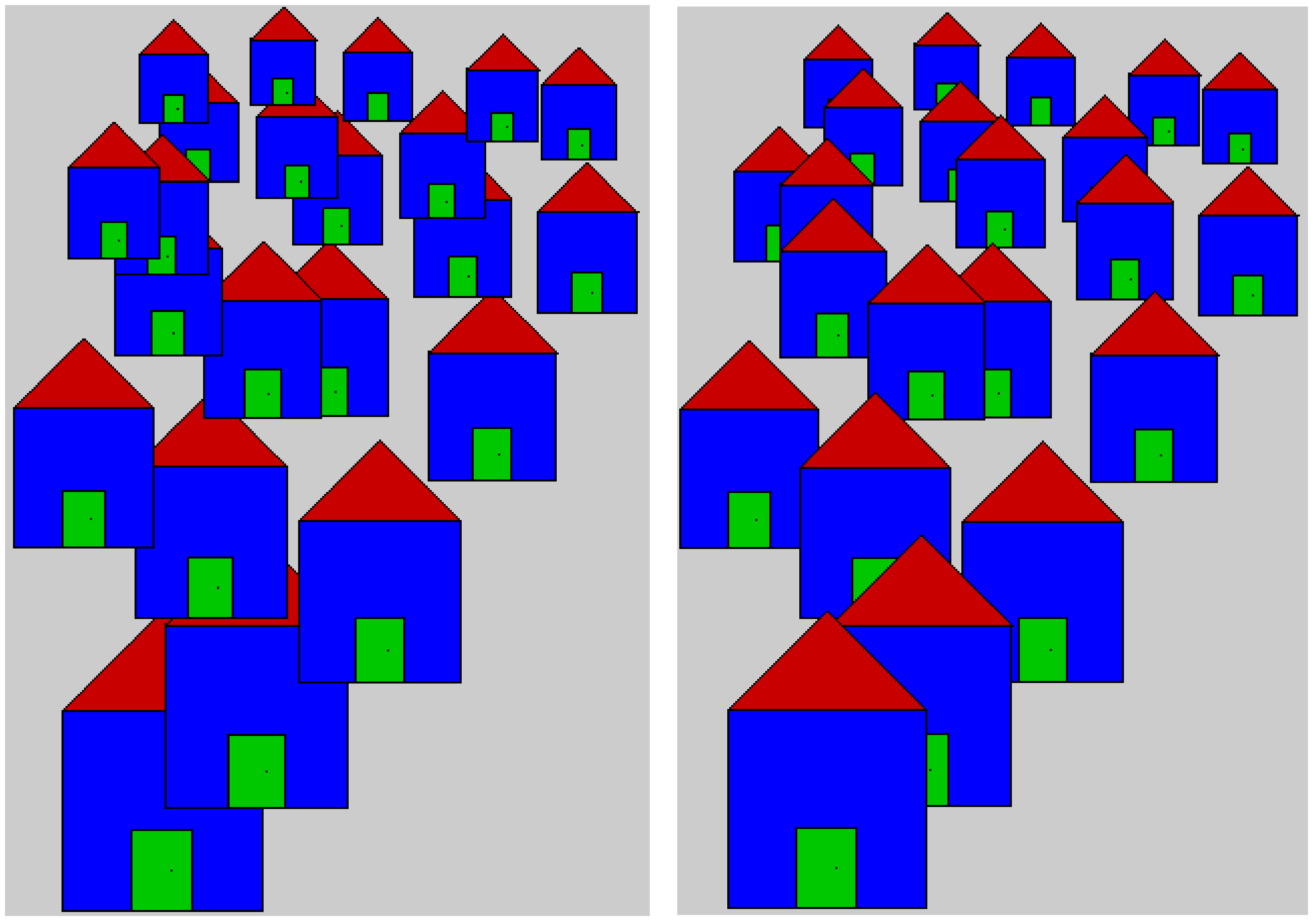
- DVDs are sorted by category at the **video store** (e.g., comedy, drama, new releases) so that we can easily find what we are looking for.

- A pile of many **trading cards** can be sorted in order to make it easier to find and remove the duplicates.

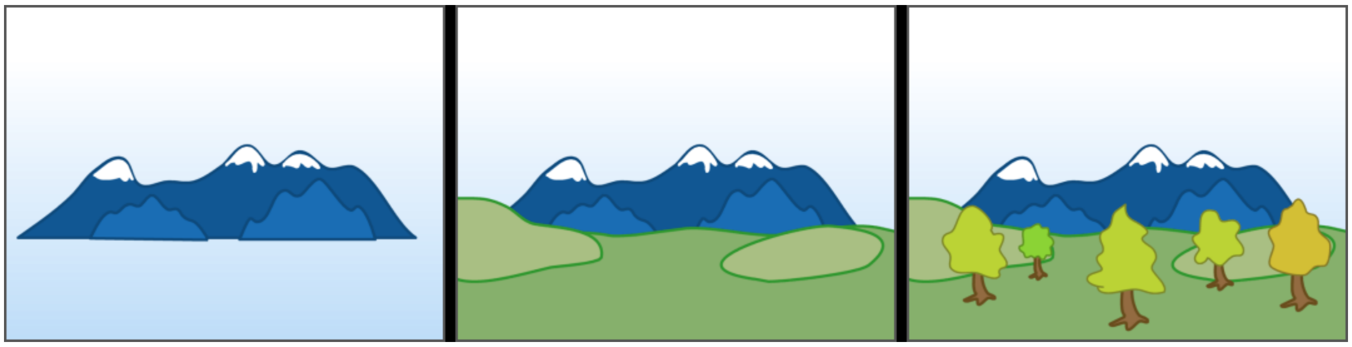
- Incoming **emails** are sorted by date so that we can read and respond to them in order of arrival.

- **Ballots** can be sorted so that we can determine easily who had the most votes.

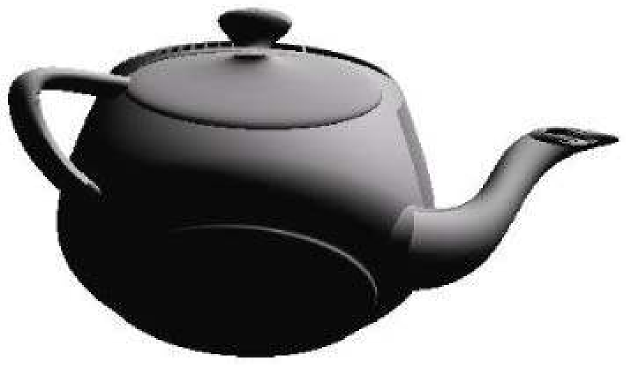
Sorting is also an important tool in computer graphics. For example, scenes in a computer-generated image may draw various objects, but the order that the objects are drawn is important. Consider drawing houses as we did earlier in the course. Here is an example of drawing them in (a) the order in which they were added to the program, and (b) in sorted order from back to front:

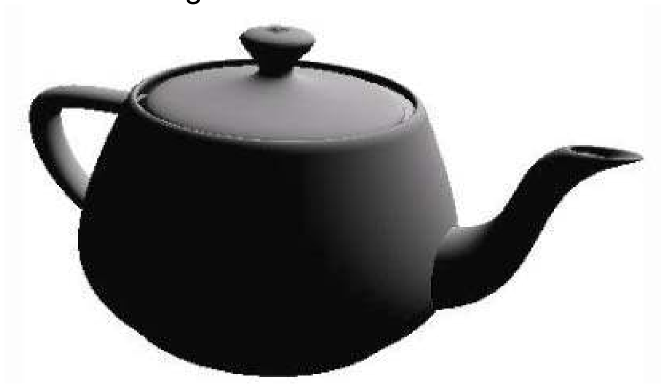

(a) normal order (b) sorted order

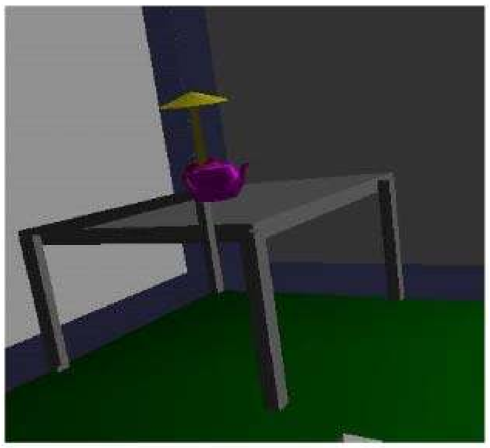
Notice how the houses in (a) are not drawn realistically in terms of proper perspective. In (b), however, the houses are displayed from back to front. That is, those with a smaller y-value (i.e., topmost houses) are displayed first. Thus, sorting the houses by their y-coordinate and then displaying them in that order will result in a proper image. This idea of drawing (or painting) from back to front is called the ***painter's algorithm*** in computer science. Painters do the same thing, as they paint background scenery before painting foreground objects:

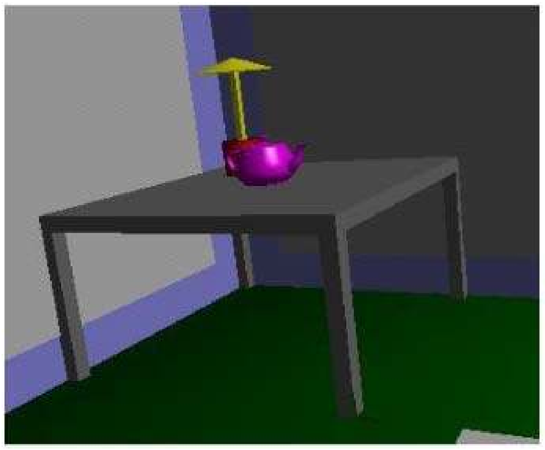


In 3D computer graphics (e.g., in a 3D game) most objects are made up of either triangular faces or quad surfaces joined together to represent 3D objects and surfaces. The triangles must be displayed in the correct order with respect to the current viewpoint. That is, far triangles need to be displayed before close ones. If this is not done correctly, some surfaces may be displayed out of order and the image will look wrong:


(a) wrong display order


(b) correct depth-sorted order


(a) wrong display order


(b) correct depth-sorted order

Thus, sorting all surfaces according to their depth (i.e., distance from the viewpoint) is necessary in order to properly render (i.e., display) a scene. The idea of displaying things in the correct order is commonly referred to as ***hidden surface removal***.

So, you can see that sorting is necessary in many applications in computer science.

Not only is sorting sometimes necessary to obtain correct results, but the ability to sort data efficiently is an important requirement for **optimizing** algorithms which may require data to be in sorted order to work correctly.

Therefore, if we can sort quickly, this can speed up search times and it can even allow our 3D games to run faster and more smoothly. Because sorting is such a fundamental tool in computer science which underlies many algorithms, many different "ways" of sorting have been developed ... each with their own advantages and disadvantages.

How many ways are there to sort ? Many.

For example, here is a table of just some types of sorting algorithms:

Sorting Style	Algorithms
Exchange Sorts	Bubble Sort, Cocktail Sort, Odd-even Sort, Comb Sort, Gnome Sort, QuickSort
Selection Sorts	Selection Sort, Heap Sort, Smooth Sort, Cartesian Tree Sort, Tournament Sort, Cycle Sort
Insertion Sorts	Insertion Sort, Shell Sort, Tree Sort, Library Sort
Merge Sorts	Merge Sort, Polyphase Merge Sort, Strand Sort
Non-Compairon Sorts	Bead Sort, Bucket Sort, Burst Sort, Counting Sort, Pigeonhole Sort, Proxmap Sort, Radix Sort

Why are there so many ways to sort ? These algorithms vary in their computational complexity. That is, for each algorithm, we can compute the (1) ***worst***, (2) ***average*** and (3) ***best*** case behavior in terms of how many times we need to compare two items from the list during the sort. Given a list of **n** items, a "good" sorting algorithm would require in the order of **n·log(n)** comparisons, while a "bad" sorting algorithm could require **n**^{2} or more comparisons.

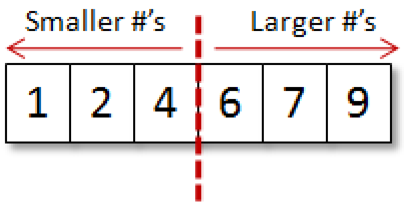
It is also possible to compare algorithms in terms of:

1. how many times a pair of items in the list are swapped (i.e., change positions).
2. how much memory (or other computer resources) is required to complete the sort.
3. how simple the algorithm is to implement.

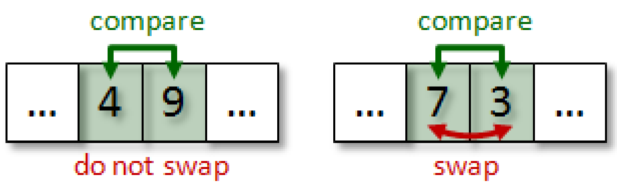
It is not the purpose of this course to make a comparison of a pile of sorting algorithms. However, it is good to get an idea as to how to write sorting routines in various ways so that you get a feel as to how various algorithms can be used to solve the same problem. We will examine a few of these now.

6.2 Bubble Sort

The **bubble sort** algorithm is easy to implement (i.e., it is easy to write the code for it). For this reason, many programmers use this strategy when they are in a hurry to write a sorting routine and when they are not worried about efficiency. The bubble sort algorithm has a bad worst-case complexity of around **n**^{2}, so it is not an efficient algorithm when **n** becomes large.
Consider sorting some integers in increasing order. The idea behind any sorting algorithm is to make sure that the small numbers are in the first half of the list, while the larger numbers are in the second half of the list:



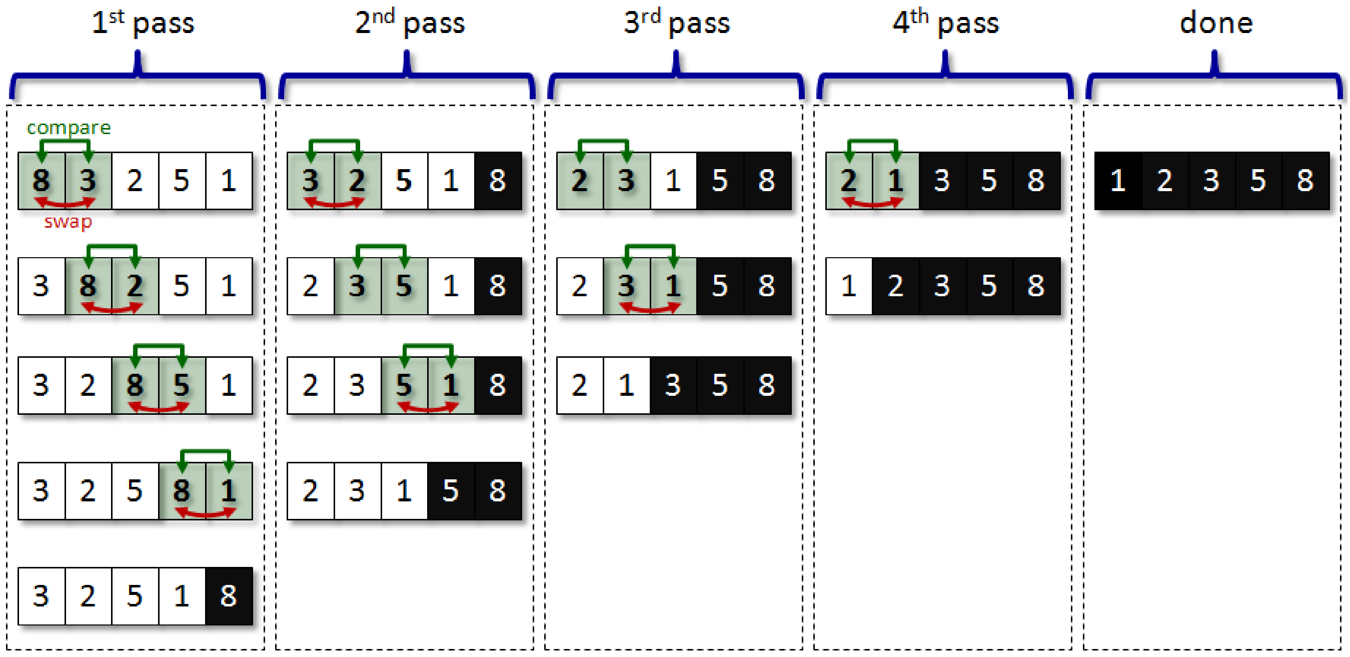
Regarding this "split" of numbers in the final sorted list, we can actually draw conclusions as to relative positions of adjacent numbers in the list as we are trying to sort. For example, assume that we examine any two adjacent numbers in the list. If the number on the left is *smaller* than the one on the right (e.g., 4 and 9 in the picture below), then relatively, these two numbers are in the correct order with respects to the final sorted outcome. However, if the number on the left is *larger* than the one on the right (e.g., 7 and 3 in the picture below), then these two numbers are out of order and somehow need to "swap" positions to be in correct order:



So, by generalizing this swapping principle to ensure "proper" ordering, we can take pairs of adjacent items in the list and swap them repeatedly until there are no more out of order. However, we need a systematic way of doing this.

The bubble sort approach is to imagine the items in a list as having a kind of "weight" in that "heavy" items sink to the bottom, while "lighter" items float (or ***bubble up***) to the top of the list. The algorithm performs the "bubbling-up" of light items by swapping pairs of adjacent items in the list such that the lighter one is ensured to be above the heavier item. It does this by making multiple passes (i.e., multiple iterations or "rounds") through the list, each time moving (or sinking) the heaviest item towards its final position at the end (or bottom) of the list.

Here is an example of how the algorithm works on a list of 5 integers:



As can be seen, during the first pass through the data, the heaviest item is moved (i.e., sinks) to the end of the list because all numbers are smaller (i.e., lighter) than it, so they "bubble-up" higher towards the top of the list. At the end of pass one, we are ensured that the largest item is at the end of the list.

During the second pass, comparisons of adjacent items are made the same way and eventually the next largest item sinks down toward the bottom. Notice that there is no need to compare the 5 with 8 at the end of the 2nd pass since the 1st pass ensured that 8 was the largest item. So the 2nd pass takes one less step to complete. The subsequent passes continue in the same manner. Once 4 passes have been completed, the list is guaranteed to be sorted.

Notice that the algorithm requires 10 comparisons of adjacent items. This is exactly twice the list size. However, as the number of items in the list grows, it is easy to see that the algorithm requires this many comparisons:

$(n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1$ ... which is: **$n \cdot (n-1)/2$** comparisons.

This is a little slow, but the algorithm is simple. Can you write the code for this algorithm ? Hopefully, you can easily see the need for nested loops, as the outer loop will cover the number of passes, while the inner loop will handle the comparisons during a single pass.

Here is a straight forward implementation:

```
Algorithm: BubbleSort1
        items:              the array containing the items to sort

1.      repeat items.length-1 times {
2.            for each location i from 0 to items.length-2 {
3.                  if (items[i] > items[i +1])  {
4.                        temp ←  items[i +1]
5.                        items[i +1] ←  items[i]
6.                        items[i] ←  temp
                    }
              }
        }
```

This implementation, however, always requires **$(n-1) \cdot (n-2)$** comparisons. We forgot to adjust the code to eliminate one less comparison in the list each time, since each pass ensures that one more item is in its final position. To do this, we need to adjust the count for the inner loop to reflect the pass number that we are making. Here is the adjusted code:

```
Algorithm: BubbleSort2
        items:              the array containing the items to sort

1.      for each pass p from items.length-1 down to 0 {
2.            for each location i from 0 to p-1 {
3.                  if (items[i] > items[i +1])  {
4.                        temp ←  items[i +1]
5.                        items[i +1] ←  items[i]
6.                        items[i] ←  temp
                    }
              }
        }
```

One more thing .... what if the list suddenly becomes sorted during the middle of the algorithm (i.e., all numbers fall into place) ? Even worse ... what if the list is *already* sorted ? We can add something to the algorithm to cause it to quit when the list is sorted. How do we know when the list is sorted ? Well ... if we go through an entire pass and did not make any swappings, then all integers must be in their correct position ... do you agree ? So we could just check to ensure that a swap was made during a pass ... and if not ... then quit:

```
Algorithm: BubbleSort3
      items:          the array containing the items to sort

1.    for each pass p from items.length-1 down to 0 {
2.          madeSwap ← false
3.          for each location i from 0 to p-1 {
4.                if (items[i] > items[i +1]) then {
5.                      temp ← items[i +1]
6.                      items[i +1] ← items[i]
7.                      items[i] ← temp
8.                      madeSwap ← true
                  }
            }
9.          if (madeSwap is false) then
                  quit()
      }
```

6.3 Selection Sort

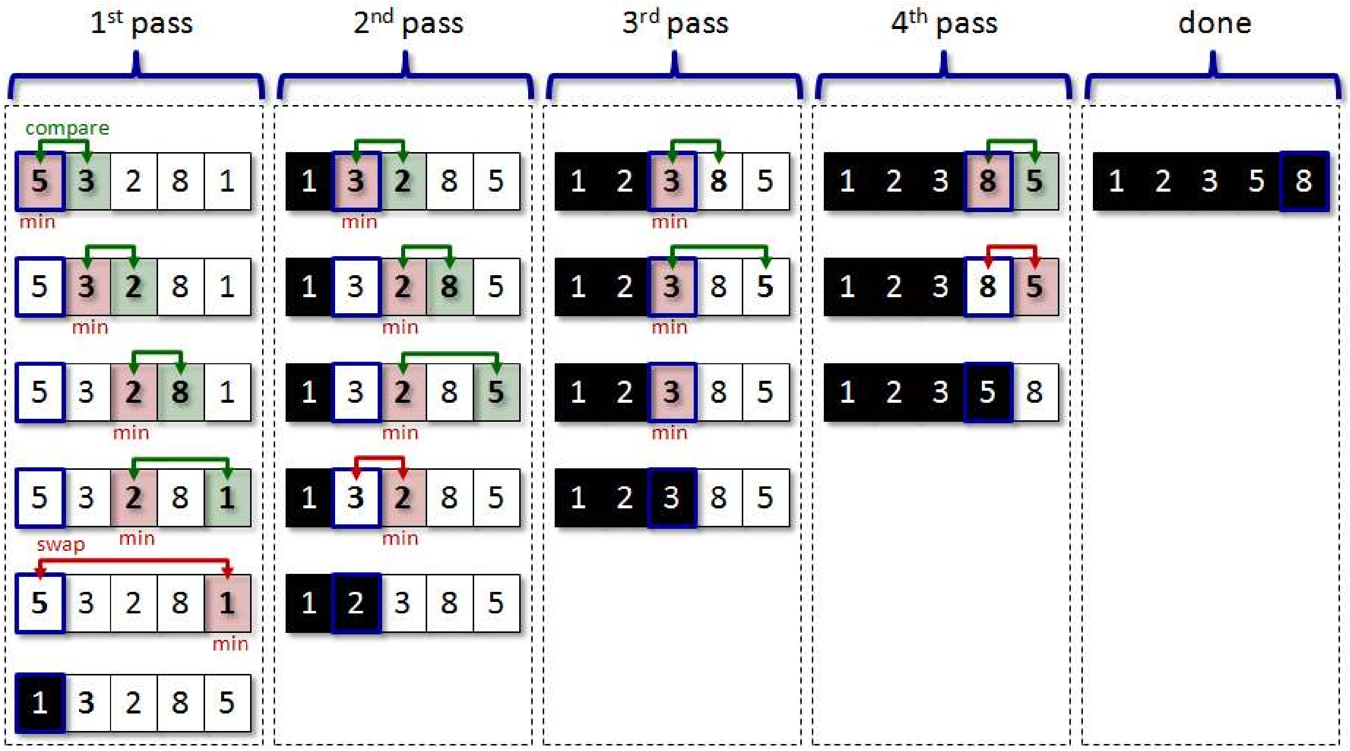
The **selection sort** algorithm is a natural kind of sorting technique. It is also easy to implement but like the Bubble Sort, it also has a bad worst-case complexity of around $\mathbf{n^2}$, so it is not an efficient algorithm when **n** becomes large.

The idea behind the algorithm is simple. It is similar to the idea of stacking a set of blocks in a tower. Find the largest block, place it at the bottom. Then find the next largest and place it on top of it, then the next largest ... and so on ... with the smallest block being placed at the top.

When the data is in a list or an array, the algorithm is slightly more complicated because we need to ensure that each item is always stored somewhere in the array. So, when we find the largest item that needs to go at the end of the array... we need to swap its position with the last item in the array. So this idea of swapping positions is necessary. The algorithm itself is usually described as finding the minimum and placing it at the front of the array ... which is opposite to the block-stacking example just mentioned, but nevertheless produces the same sorted result.

The selection sort approach is to always keep track of (i.e., hold on to) the smallest item as you go through the list. As the algorithm iterates through the items in the list it always compares against the smallest item being held onto and if a smaller item comes along, it them becomes the smallest item. Once the list has been checked, we then move the smallest item to the front of the list and do another round starting with the second item.

Here is an example of how the algorithm works on a list of 5 integers:



Notice that the algorithm again requires 10 comparisons of adjacent items as well as 3 swaps. As with the bubble sort, it is easy to see that the selection sort may require **n·(n-1)/2** comparisons. However, much less swaps are made with the selection sort. The selection sort may require **(n-1)** swaps, while the bubble sort can require up to **n·(n-1)/2** (e.g., when the list is in reverse order). You can see therefore, that the selection sort is a little more efficient.

The code for this algorithm is quite similar to that of the bubble sort in that it has nested loops and compares items. However, this time, we compare each item against the minimum, not against its adjacent neighbor. Also, the swap occurs outside the inner loop... not within the inner loop as with the bubble sort.

Here is a straight-forward implementation:

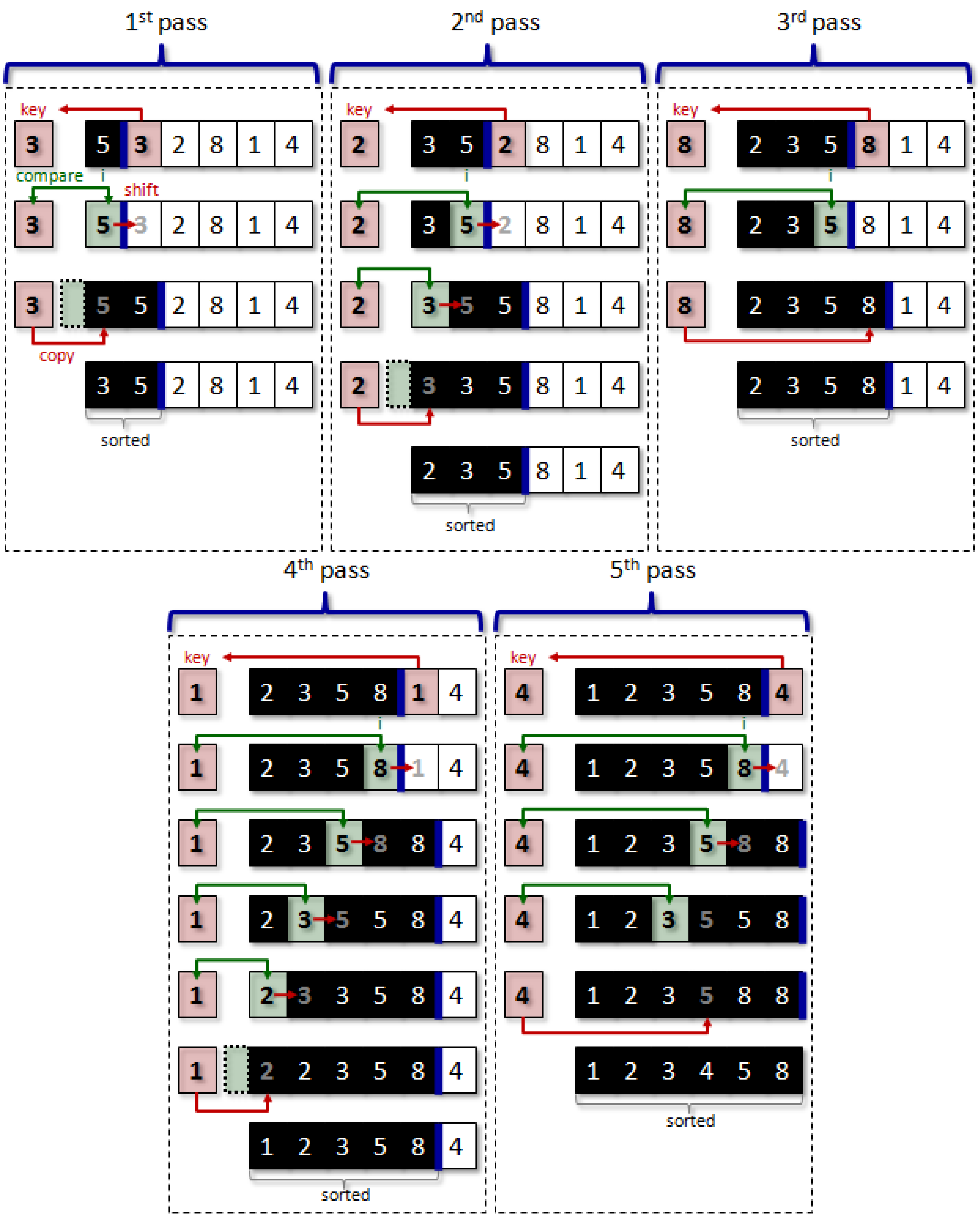
```
Algorithm: SelectionSort
        items:            the array containing the items to sort

1.      for each pass p from 0 to items.length-1 {
2.            minIndex ← p
3.            for each index i from p+1 to items.length-1 {
4.                  if (items[i] < items[minIndex]) then
5.                        minIndex ← i
              }
6.            temp ← items[p]
7.            items[p] ← items[minIndex]
8.            items[minIndex] ← temp
        }
```

6.4 Insertion Sort

The **Insertion Sort** algorithm is perhaps the most natural sorting algorithm. It is very much like the Selection Sort in that it attempts to sort by selecting one item at a time. Even though it has a bad worst-case complexity of around $\mathbf{n}^2$, in general it is faster than the Selection Sort because it does not search the whole list to look for the smallest item first.

The idea behind the algorithm is simple and relates to a real-life kind of sort that we would naturally perform. Imagine on the table a "pile" of unsorted items. The idea is to repeatedly select items from the unsorted pile and place them in a newly sorted pile. So, the algorithm maintains a portion of items that are sorted (i.e., the ones at the front of the list) and a portion of items that still remain to be sorted (i.e., the ones at the back of the list). Each time an item is selected, the "sorted portion" grows by one, while the "unsorted pile" shrinks by one. After doing this **n** times, the whole list is sorted. It is similar to the idea of a librarian placing books on a shelf with "already-sorted" books.

Here is an example of how the algorithm works on a list of 6 integers:



Notice how the left side of the list always contains items in sorted order, although additional items still need to be inserted in there, so that sorted list is not complete until the last pass. Notice in all except the 3rd pass that there was a need to search backwards through the sorted portion in order to find the correct place to insert the **key** item.

Here is a straight-forward implementation. Notice the use of a **while** loop in order to allow the loop to exit quickly as soon as the **key** item is larger than the ones remaining in the sorted portion of the list:

```
Algorithm: InsertionSort
        items:              the array containing the items to sort

1.      for each pass p from 1 to items.length-1 {
2.            key ← items[p]
3.            i ← p - 1
4.            while (i >= 0) AND (items[i] > key) {
5.                  items[i +1] ← items[i]
6.                  i ← i - 1
              }
7.            items[i +1] ← key
        }
```

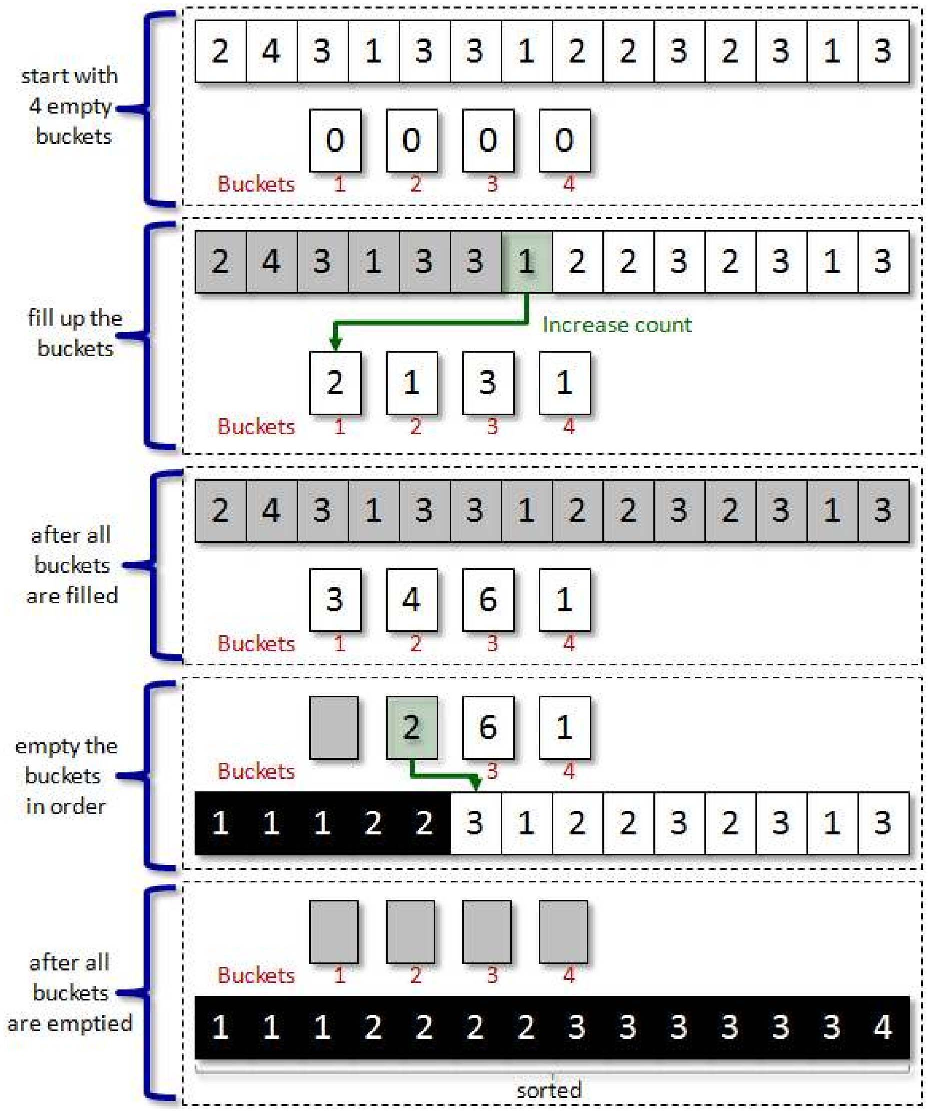
With a careful look at the code, you can see that the algorithm may need to make **n·(n-1)/2** comparisons as with the Selection Sort. Also, there may be a need to make this many swaps as well. However, in a typical scenario with an initial random arrangement of numbers, the Insertion Sort takes about half the speed of a Selection Sort ... due to the ability of the **while** loop to exit earlier. So, in general, the Insertion Sort is a little more efficient.

6.5 Bucket Sort & Counting Sort

The **Bucket Sort** algorithm is an excellent sorting algorithm for the special case in which there are many duplicate items and the items are bounded by some small maximum size. It is a simple algorithm and can run in worst-case complexity of **2·n** time (under certain situations) !

The idea of the algorithm is similar to that of what you may find at the post office. Incoming mail is quickly placed into "roughly sorted" bins (or buckets). Each item in a particular bin is "equal" in some sense of the word (e.g., same destination city, same postal code, same street, etc..). So each bin represents a partially-sorted list. Each bin can then be sorted separately, in any manner. A special case arises when multiple items are considered equal. That is, consider 1,000 student exam papers with integer grades ranging from 0% to 100% that need to be sorted by grade. You can sort them by making 101 bins representing the grades and then placing each exam in the corresponding bin according to the grade.

In the case where we actually have a set of fixed-range integers that we need to store, the algorithm becomes what is known as a **Counting Sort** and is quite simple. Consider an array with 14 numbers as show below. The array is assumed to contain integers from 1 to 4. We can make 4 buckets in the form of integer counters and then simply fill up the corresponding bucket counter as we iterate through the numbers. Then, to get the sorted list, we just empty the buckets in order →


start with 4 empty buckets
fill up the buckets
Increase count
after all buckets are filled
empty the buckets in order
after all buckets are emptied
Buckets
sorted

This is very simple and it only takes **2n** steps. Here is a straight-forward implementation:

Algorithm: BucketSort1

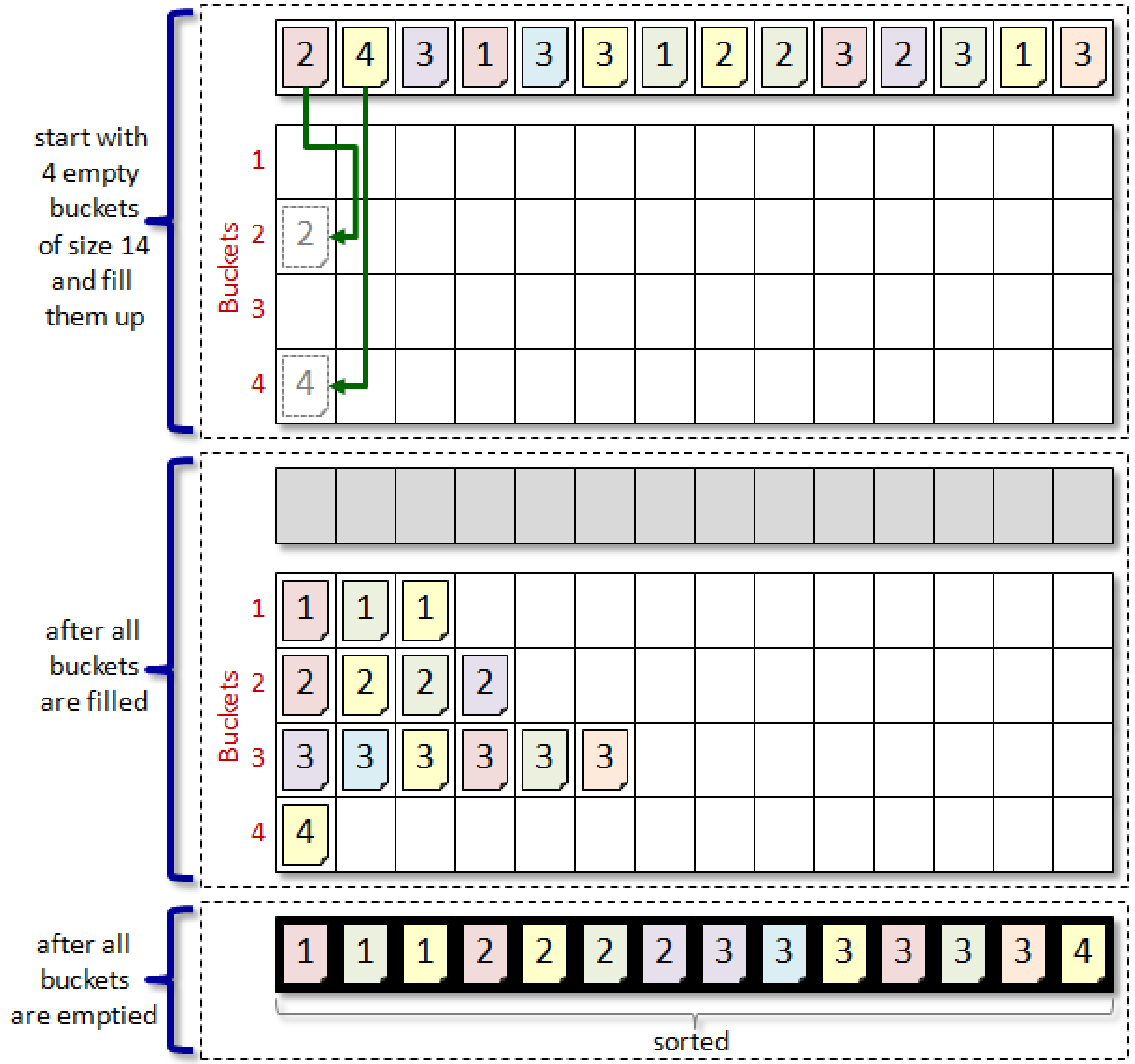
```
        items:          the array containing the items to sort
        b:              the number of bins to use

1.      bins ← new array of size b each element set to 0

2.      for each pass i from 0 to items.length-1 {
3.          bins[items[i]-1] ← bins[items[i]-1] + 1
        }

4.      i ← 0
5.      for each bin x from 0 to b-1 {
6.          for each count c from 0 to bins[x] {
7.              items[i] ← x + 1
8.              i ← i + 1
            }
        }
```

Notice in steps 2 and 3 how the bins array simply stores a count of how many items had that bin's value. Then in steps 4 through 8 we simply go through each bin and fill in the items array with the correct number of items from the bins.

A slightly more complex situation arises when we actually need to store the items themselves in the bucket (i.e., not just counters). In our example with the exam papers, we need to store the exam papers themselves, not just the grades.

To accomplish this, we need to store more than a counter in each bin. In fact, we need to reserve space for the items themselves. How many items may fit into a bin ? Well, if all the items are equal, they will end up in the same bin!! Therefore, even though it is likely that a typical random set of items will be distributed evenly among the bins, it is possible that some bins may get very full. So, to be safe, we would need to make each bin large enough to hold all the items.

Therefore, our bins array in the above code would need to be a two-dimensional array of size **b·n** so that each of the **b** buckets can store up to **n** items. Here is what the algorithm will do:


start with 4 empty buckets of size 14 and fill them up
2 4 3 1 3 3 1 2 2 3 2 3 1 3
Buckets
1
2
3
4
after all buckets are filled
1 1 1
2 2 2 2
3 3 3 3 3 3
4
after all buckets are emptied
1 1 1 2 2 2 2 3 3 3 3 3 3 4
sorted

Now, in reality, only **n** items are being stored in the buckets. Therefore **(b-1)·n** spaces in the 2D array will remain empty. This is a little bit wasteful.

In our exam paper example, the 2D array would need to have space to store 101·1000 = 101,000 exam papers, although only 1,000 exam papers would actually be stored!!! That is about 99% of wasted space! Of course, there are ways to fix this, as you will learn in your 2nd year here in computer science. We can, for example, make initially small buckets based on "estimates" as to how many items we expect to fall into any given bucket ... and then *grow* the buckets as they become full. We will not discuss this further here. However, be aware that there is often a trade-off between runtime complexity and storage space.

Here is the adjusted code to handle the storage of items instead of counters:

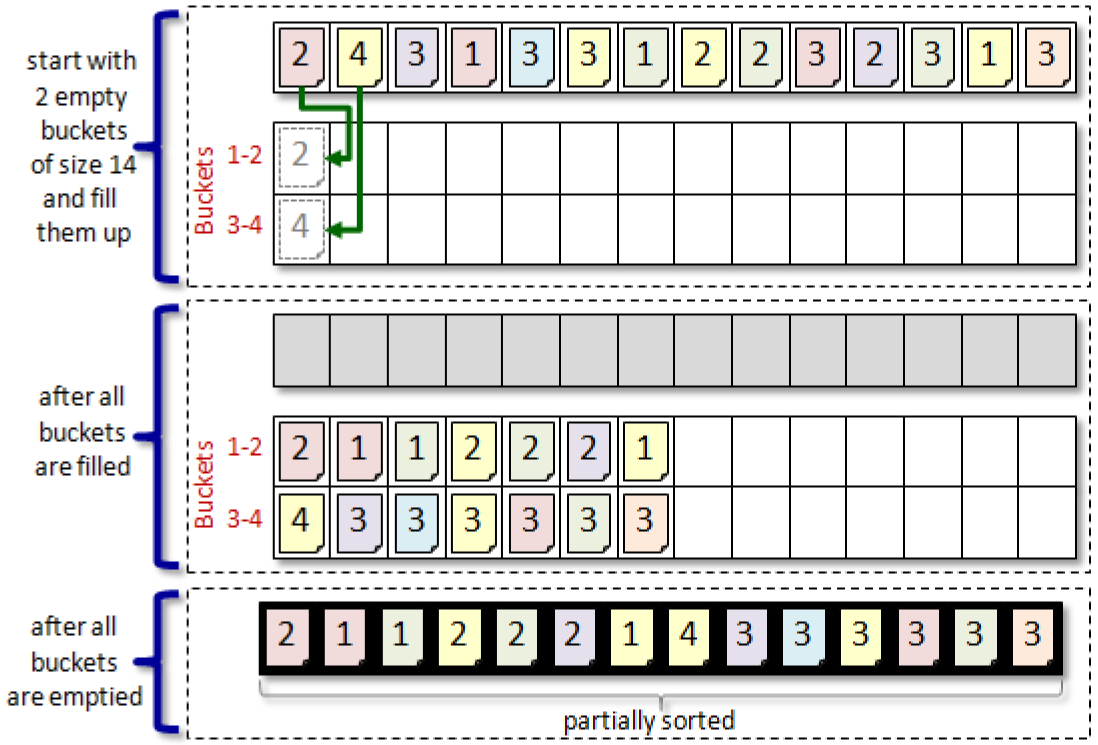
```
Algorithm: BucketSort2
        items:          the array containing the items to sort
        b:              the number of bins to use

1.      bins ← new array of b empty arrays each of size items.length
2.      binCount ← new array of b counters initially set to 0

3.      for each pass i from 0 to items.length-1 {
4.            binID ← getBinFor(items[i])
5.            bins[binID][binCount[binID]] ← items[i]
6.            binCount[binID]← binCount[binID] + 1
        }
7.      i ← 0
8.      for each bin binID from 0 to b-1 {
9.            for each item c from 0 to binCount[binID] {
10.                 items[i] ← bins[binID][c]
11.                 i ← i + 1
              }
        }
```

The above code assumes that the order of any two exam papers with the same grade is arbitrary/unimportant and thus do not need to be re-arranged in any way.

However, the algorithm can be generalized by allowing less bins. For example, considering our exam paper example, if we use only 10 bins instead of 101, this would significantly reduce the storage requirements.

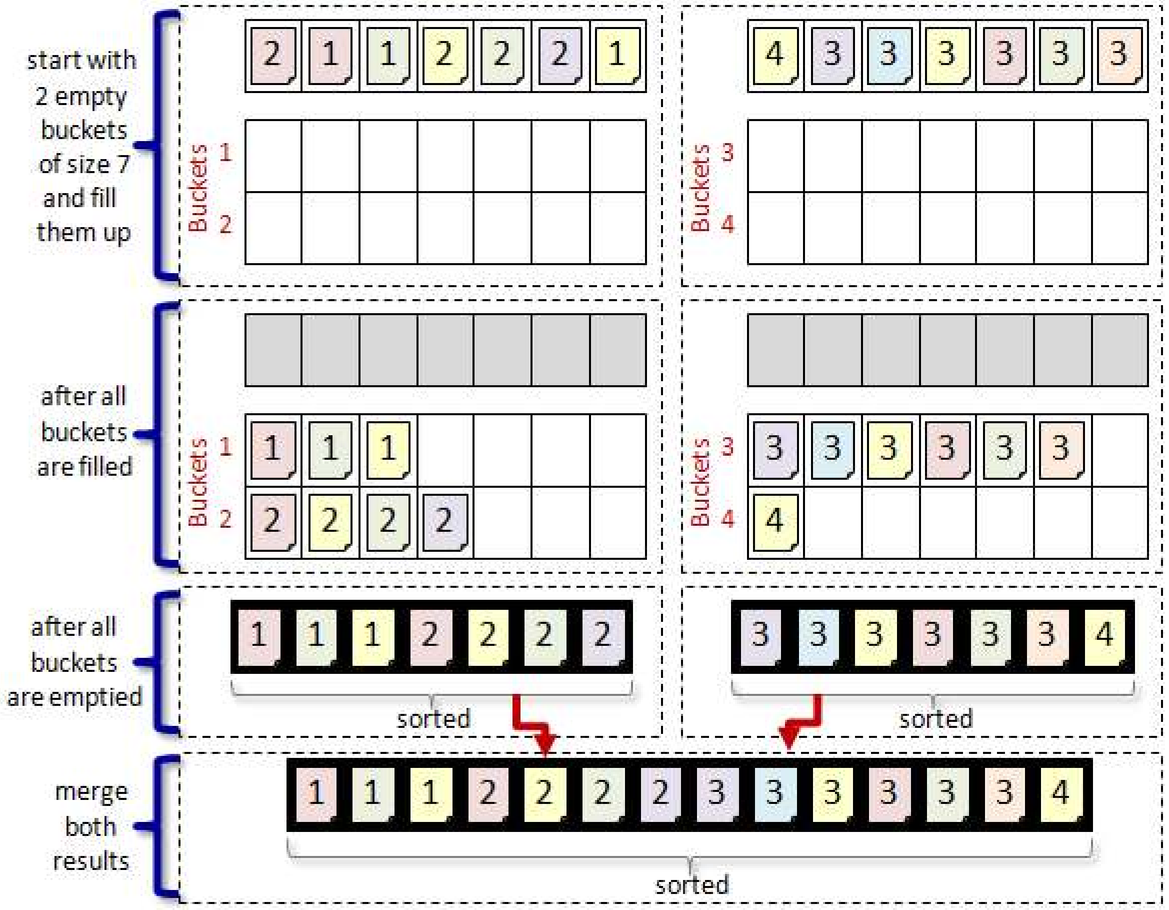
We could, for example, put all grades in the 70%-79% range into one bin, all grades in the 80%-89% range into another bin, etc..

The result is that the array would be partially sorted, but not complete. Here is our example again but with 2 buckets instead of 4:


start with 2 empty buckets of size 14 and fill them up
2 4 3 1 3 3 1 2 2 3 2 3 1 3
Buckets
1-2
3-4
2
4
after all buckets are filled
2 1 1 2 2 2 1
4 3 3 3 3 3 3
after all buckets are emptied
2 1 1 2 2 2 1 4 3 3 3 3 3 3
partially sorted

Notice that there is much less wasted space, but the end-result is that the array is not sorted. A solution would be to sort each bucket before filling up the array again. Depending on the size, we could use any sorting technique to sort the buckets.

We could, for example, use a bucket sort again on the two buckets:



The actual merging of the sorted buckets is easy again, as we simply use the same code as before. The speed of the algorithm will depend on the kind of sorting technique that we use.

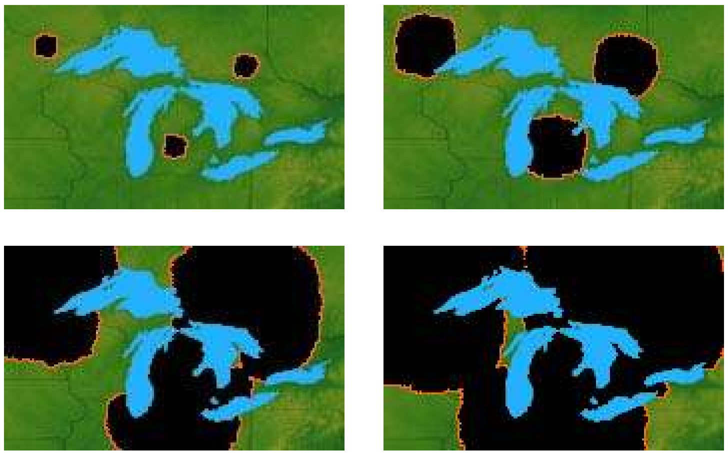
Here is the code for the general **Bucket Sort** where the buckets need to be sorted:

```
Algorithm: BucketSort3
        items:          the array containing the items to sort
        b:              the number of bins to use

1.      bins ← new array of b empty arrays each of size items.length
2.      binCount ← new array of b counters initially set to 0

3.      for each pass i from 0 to items.length-1 {
4.          binID ← getBinFor(items[i])
5.          bins[binID][binCount[binID]] ← items[i]
6.          binCount[binID]← binCount[binID] + 1
        }
7.      for each bin binID from 0 to b-1 {
8.          sort(bins[binID])        // use any algorithm...will affect runtime though
        }
9.      i ← 0
10.     for each bin binID from 0 to b-1 {
11.         for each item c from 0 to binCount[binID]  {
12.             items[i] ← bins[binID][c]
13.             i ← i + 1
            }
        }
```

In summary, with a large number of items and enough bin space (i.e., storage space), then a **Counting Sort** is the best that we can hope for since it minimizes the number of steps needed to be made in order to sort. However, remember that it only works well if there are a lot of items that are equal. The more general **Bucket Sort** is used when the bin size is greater than one and this can also be very efficient when there are many equal items.

6.6 An Example - Fire Spreading Simulation

Consider another example of where sorting is necessary. Assume that we want to simulate the spread of a fire across a terrain. Here is an example that shows three separate fires spreading across a forested area, leaving charred remains behind:



To make all of this happen, we first need to understand how the forest & lakes are represented, stored and displayed in our simulation. The following processing code displays an image of forest and lakes from a file called "smallLakes.png":

```
PImage     terrain;   // image representing the forests and lakes

void setup() {
  terrain = loadImage("smallLakes.png");
  size(terrain.width, terrain.height);
  image(terrain, 0, 0);
  loadPixels();
}

void draw() {
  updatePixels();
}
```

In the code above, the **terrain** variable is of type **PImage** which is the data type for storing images in Processing. You must call the **loadImage()** function before you can use the image, supplying the name of the image file that you want to load (which must be of type **.gif**, **.jpg**, **.tga**, or **.png**).

Once loaded, you can access the **PImage** object which contains fields for the **width** and **height** of the image, as well as an array called **pixels[]** which contains the values for every pixel in the image. You can use the **width** and **height** values to decide how big to make your window.

The **image(anImage, x, y)** procedure will draw **anImage** onto the window with the top/left of the image starting at the given **x** and **y** coordinate on the window (although in Processing there is a minimum window width & height so if the image is very small, it may be centered in the window).

Once the image has been displayed, you can call the **loadPixels()** procedure to load the pixel data from the display window into a pre-defined Processing array called **pixels**. The **loadPixels()** procedure must always be called before reading from or writing to **pixels**. We can use the pixels array to examine and modify any pixel in the image (i.e., to determine whether or not it is grass (green) or water (blue) and to set it to show a fire (orange) or a burnt area (black).

Finally, the **updatePixels()** procedure will take the (possibly modified) data from the **pixels** array and draw the image again on the window. During our simulation, we will do this repeatedly in the draw procedure to show visually that the fires are spreading.

So, how then do we represent the fires ? Well, a fire can be shown by simply changing a pixel to orange. Say, for example that we wanted the pixel at location (85,70) in the image to be on fire. We simply need to change the correct pixel in the **pixels** array to orange. Since the pixels array is one-dimensional, we need to determine the index in the array that represents (85,70) in the image.

The **pixels** array stores each row of the image one after another. So the first row has **terrain.width** pixels in it and these are the first pixels in the array representing the pixels in which y = 0. Hence, if we want the 71st row (i.e., y = 70, since y starts at 0), then we need to multiply the width of the image by the row (i.e., 70) to bypass the first 70 rows.

Therefore, to calculate the index of (85,70) we need to use this formula:

```
pixels[85 + (70*terrain.width)]
```

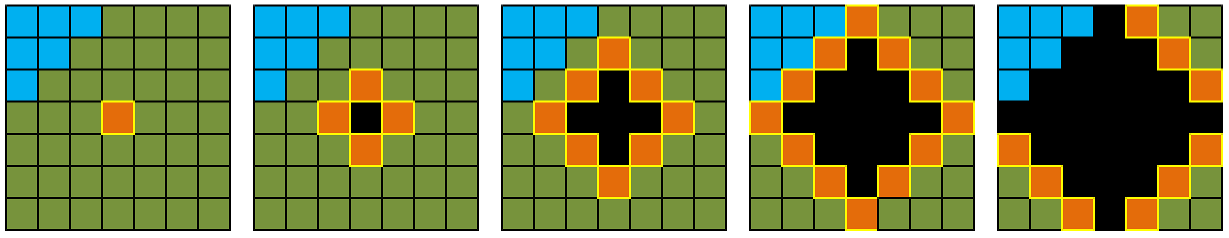
In general, the pixel at position (x, y) in the image can be set to orange as follows:

```
pixels[x + (y*terrain.width)]  = color(255, 100, 0);   // lotsa red & a bit of green = orange
```

So, we can simulate the fires by changing the appropriate pixels to orange for a while and then black afterwards to indicate a burned area.

But to simulate the fire spreading process properly, we need to make the fires grow outwards from their starting locations. But how can we do this ?

Intuitively, green pixels close to the starting location need to "catch fire" before ones that are further away. Here, for example is a single fire spreading outwards:


(a) (b) (c) (d) (e)

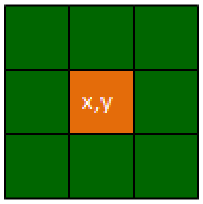
Notice that as the fire spreads ... there is a "wall" of fire that expands outwards, while the interior burned portions remain black. Intuition again tells us that we only need to consider this "wall" (also known as the ***active border***) of the fire in order to determine the spread-pattern of the fire. That is, each orange pixel is a miniature fire that needs to be spread outwards.

Therefore, we will need to maintain this active border of all locations that are currently on fire so that we can propagate the fire outwards from each of these locations. These border locations can be simply represented as (x,y) points and to begin ... we can add the first fire pixel location as the only border point upon startup. Here is how we can add code to do this:

```
PImage      terrain;       // The image with forest and lakes
Point       start;         // start location of fire
Point[]     border;        // points along active border of fire
int         borderSize;    // # points along active border of fire

// Data structure to represent a point
class Point {
  int x, y;
  Point(int ix, int iy) {
      x = ix;
      y = iy;
  }
}

void setup() {
  terrain = loadImage("smallLakes.png");
  size(terrain.width, terrain.height);
  image(terrain, 0, 0);
  loadPixels();

  border = new Point[1000];
  border[0] = new Point(85,70);
  borderSize = 1;
}
```

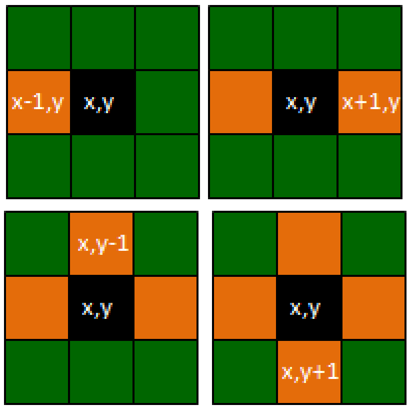
Notice that the border array begins with one point, but that it is able to hold a lot more as the border grows. The number 1000 is somewhat arbitrary, but it is big enough to hold a lot of border points. If the entire 107x102 pixel image was on fire ... that would require 17,340 border pixels potentially. However, we must remember that as the border grows it does not get very large since the previous border points will change to black as the fire consumes the forest. So, 1000 points is reasonable in our example, although with larger images a larger border would be necessary.

So now that we know the active border pixels, how do we "grow" the fires ? Well, for each pixel on the active border, we just need to look at the neighboring pixels surrounding it (i.e., see picture (b) on the previous page). Assume that the pixels are stored in a 2-dimensional image called **image** and that each location **image[x][y]** is either colored green, blue, orange or black.



Given that a **image[x][y]** is on the active border of the fire, here is the idea for spreading the fire from point (x,y):

```
image[x][y] ← BLACK      // make burnt now

image[x-1][y] ← ORANGE   // burn left now

image[x+1][y] ← ORANGE   // burn right now

image[x][y-1] ← ORANGE   // burn up now

image[x][y+1] ← ORANGE   // burn down now
```



We will need to make sure that all of these "new" fires are added to the active border for the next round of spreading. So, we need to repeat the process of extracting a fire location from the active border, processing it (i.e., spread outwards from here) and then repeat.

Here is the algorithm that we have so far now:

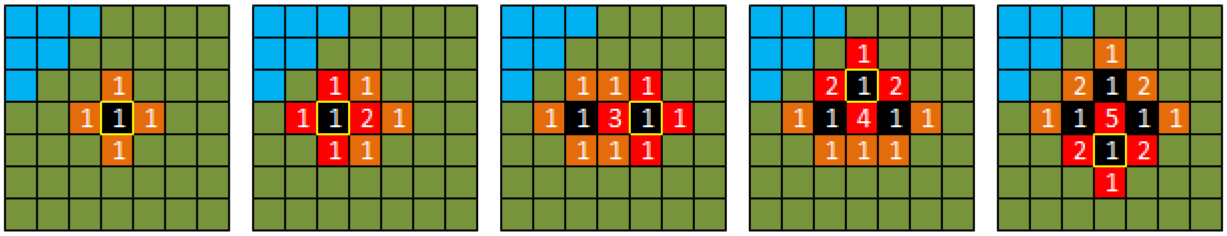
```
Algorithm: FireSpread
      image:   the image containing forests and lakes
      start:   point representing the starting location of the fire

1.    borders = new array of points, initially empty
2.    borderSize = 0
3.    borders[0] = start
4.    while (borderSize > 0) {
5.        x ← borders[borderSize-1].x
6.        y ← borders[borderSize-1].y
7.        image[x][y] ← BLACK                    // burnt
8.        borderSize ← borderSize - 1

9.        borders[borderSize] ← new Point (x-1, y)
10.       borderSize ← borderSize + 1
11.       image[x-1][y] ← ORANGE

12.       borders[borderSize] ← new Point (x+1, y)
13.       borderSize ← borderSize + 1
14.       image[x+1][y] ← ORANGE

15.       borders[borderSize] ← new Point (x, y-1)
16.       borderSize ← borderSize + 1
17.       image[x][y-1] ← ORANGE

18.       borders[borderSize] ← new Point (x, y+1)
19.       borderSize ← borderSize + 1
20.       image[x][y+1] ← ORANGE
      }
```

We need to make sure however, that we do not process any pixels outside the valid range. For example, if the border point is the top left pixel in the image, then we cannot try to propagate the fire upwards nor leftwards because the point (-1,-1) would be out of range. Therefore, we need to add some "bounds-checking" to ensure that this does not happen.

We need to place conditional statements around lines 9-11, 12-14, 15-17 and 18-20 as follows:

```
9.      if (x > 0) then {                  // make sure not gone beyond left border
10.         borders[borderSize] ← new Point (x-1, y)
11.         borderSize ← borderSize + 1
12.         image[x-1][y] ← ORANGE
        }

13.     if (x < image.width-1) then {      // make sure not gone beyond right border
14.         borders[borderSize] ← new Point (x+1, y)
15.         borderSize ← borderSize + 1
16.         image[x+1][y] ← ORANGE
        }
17.     if (y > 0) then {                  // make sure not gone beyond top border
18.         borders[borderSize] ← new Point (x, y-1)
19.         borderSize ← borderSize + 1
20.         image[x][y-1] ← ORANGE
        }
21.     if (y < image.height-1) then {     // make sure not gone beyond bottom border
22.         borders[borderSize] ← new Point (x, y+1)
23.         borderSize ← borderSize + 1
24.         image[x][y+1] ← ORANGE
        }
    }
```

At this point, we still have a slight problem. When we extract a border point, we need to ensure that we don't re-propagate through the same points over and over again (i.e., through points that are ORANGE or BLACK), otherwise we will never have an end to our fires and fires can restart. Below, for example, shows how we spread out from the 4 fires around the center, showing the number of times a location is added as a fire to the active border:



It may come as a surprise to you that 5 separate fires end up starting at the center location ... even after that location has been burned. We need to ensure that we do not add points to the border if there is already a fire there (i.e., ORANGE) or if there is a burned area there (i.e., BLACK) or if there is water there (i.e., BLUE).

To do this, we need to again modify lines 9, 13, 17 and 21 as follows:

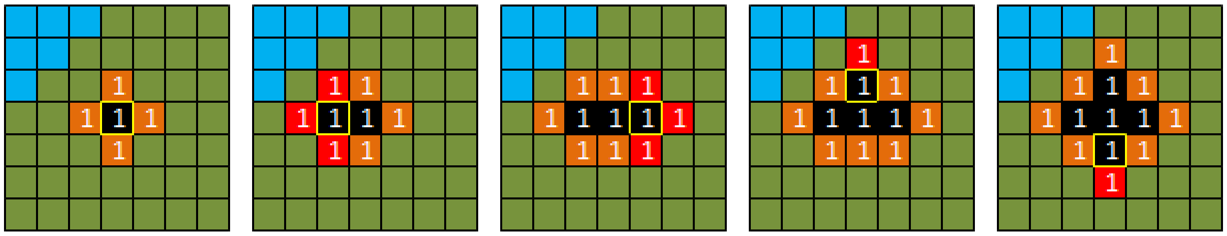
```
9.      if ((x > 0) AND (image[x-1][y] is GREEN)) then {
```

```
13.     if ((x < image.width-1) AND (image[x+1][y] is GREEN)) then {
```

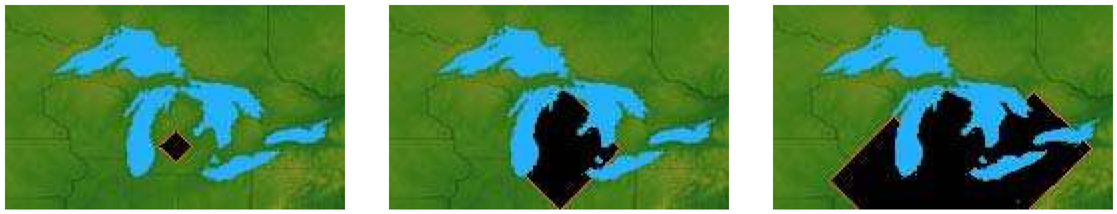
```
17.     if ((y > 0) AND (image[x][y-1] is GREEN)) then {
```

```
21.     if ((y < image.height-1) AND (image[x][y+1] is GREEN)) then {
```

As a result, we do not repeatedly add fires to the same location:



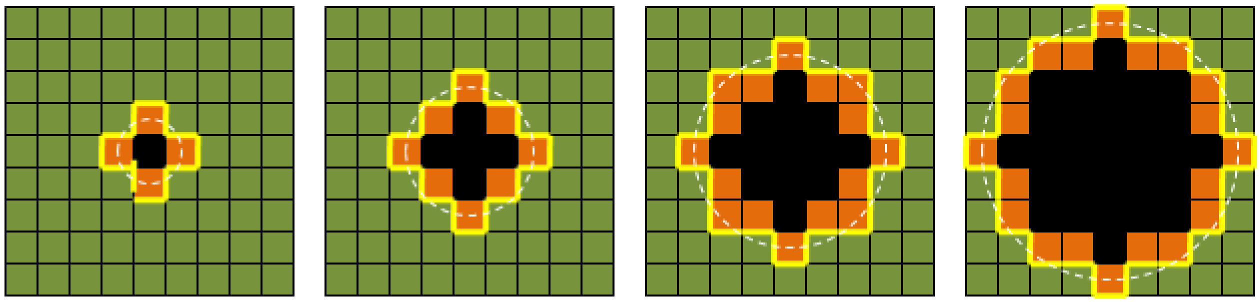
At this point, the active border will begin by growing from 1 point to having 4 new points in it, representing 4 new fires (as shown above). However, you may notice in the image above that the fire will spread in a diamond-like pattern. Here is what it would look like as we continued this process:



This pattern is called an *artifact* of the simulation.

> *An **artifact** is something that appears in a scientific result that is not a true feature of thing being studied, but instead a result of the experimental or analysis method, or observational error.*

Clearly, this is not a realistic fire-spread model as fires do not spread in diamond-shape formation. In order to make the spreading more realistic, we need to cause the fire to spread outwards by processing the pixels outwards in a circular pattern from the starting fire location:



To do this, we need to ensure that active border points that are closest to the center are processed first. So, we need to sort the border points by their distance to the starting location of the fire.

One way of doing this is to compute the distance from the center of the fire (i.e., the initial starting location) to each point on the active border. Then, we can sort the points on the active border according to their distance to the fire center and make sure that the next point to be processed is the one that is closest to the center.

To do this, we would need to keep track of the distance to each point on the active border which would be set to the distance from the starting fire location. To represent these distances, we can have each point on the active border maintain its distance from the fire's center. In processing, we could re-define the Point data structure as shown here. Here are the changes to the algorithm:

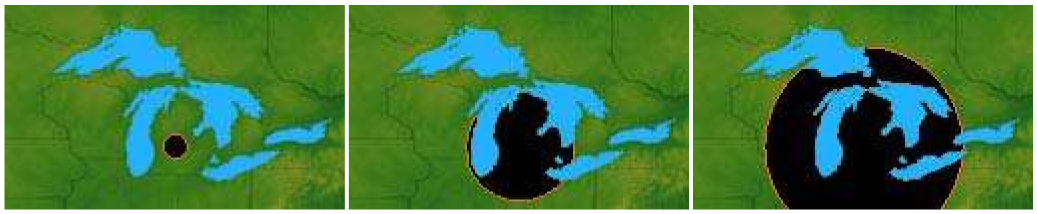
```
class Point {
  int x, y, distance;
  Point(int ix, int iy, float d) {
      x = ix;
      y = iy;
      distaince = d;
  }
}
```

Algorithm: FireSpread(image, start)

```
1.  borders = new array of points, initially empty
2.  borderSize = 0
3.  borders[0] = start
4.  while (borderSize > 0) {
5.      x ← borders[borderSize-1].x
6.      y ← borders[borderSize-1].y
7.      image[x][y] ← BLACK
8.      borderSize ← borderSize - 1
9.      if ((x > 0) AND (image[x-1][y] is GREEN)) then {
10.         borders[borderSize] ← new Point (x-1, y, distance from start to (x-1, y))
11.         borderSize ← borderSize + 1
12.         image[x-1][y] ← ORANGE
        }
13.     if ((x < image.width-1) AND (image[x+1][y] is GREEN)) then {
14.         borders[borderSize] ← new Point (x+1, y, distance from start to (x+1, y))
15.         borderSize ← borderSize + 1
16.         image[x-1][y] ← ORANGE
        }
17.     if ((y > 0) AND (image[x][y-1] is GREEN)) then {
18.         borders[borderSize] ← new Point (x, y-1, distance from start to (x, y-1))
19.         borderSize ← borderSize + 1
20.         image[x][y-1] ← ORANGE
        }
21.     if ((y < image.height-1) AND (image[x][y+1] is GREEN)) then {
22.         borders[borderSize] ← new Point (x, y+1, distance from start to (x, y+1))
23.         borderSize ← borderSize + 1
24.         image[x][y+1] ← ORANGE
        }
25.     sort borders by distances from largest to smallest
    }
```

Any sort algorithm will suffice, as long as the sorting is done with respect to the distances stored in each border point.

The above code will produce a nice "round" fire spread ... however ... the fire is still not spreading realistically. Can you see what is wrong in the following images:



Notice particularly the 2nd image. See how the fire quickly spreads around the lake on the left side of the fire border circle ? This is not realistic as the fire must spread from the bottom left side of the lake upwards....and this takes time. So, the point is... we cannot assume that the fire should be processed with respect to the distance from the center of the fire's starting point. Instead, we must adjust the code so that the fire "bends" properly around obstacles.

To do this, we need to adjust our computation so that each new fire spreads time-wise relative to the fire that spawned it. That is, the cost of a fire should be with respect to the distance from the border point that it started from. Therefore, we need to adjust lines 10, 14, 18 and 22 as follows:

```
10.        borders[borderSize] ← new Point (x-1, y, distance from (x, y) + 1)
```

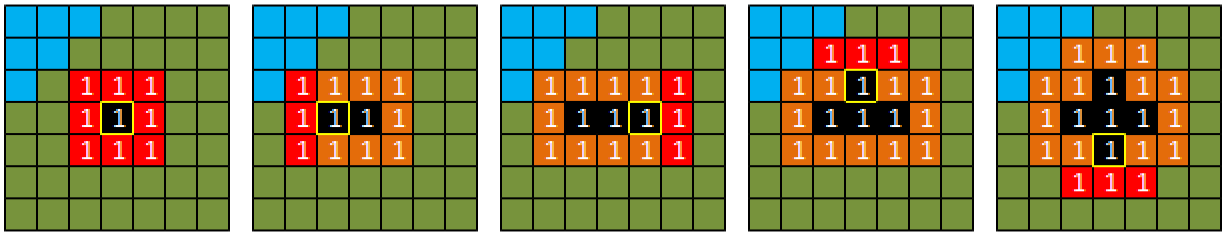
```
14.        borders[borderSize] ← new Point (x+1, y, distance from (x, y) +1)
```

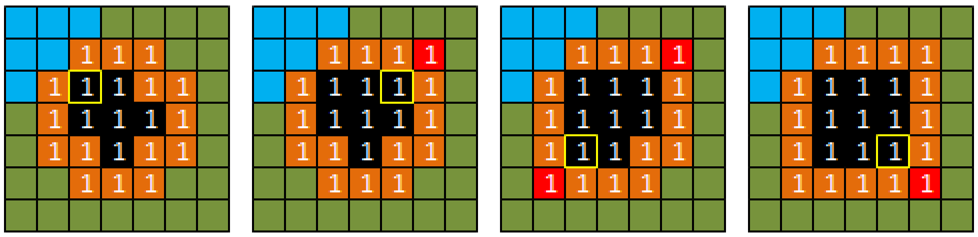
```
18.        borders[borderSize] ← new Point (x, y-1, distance from (x, y) + 1)
```

```
22.        borders[borderSize] ← new Point (x, y+1, distance from (x, y) + 1)
```

The above code allows each new fire point to have a distance value that is 1 more than the previous distance value.

However, oddly enough, the code will still produce a result that has a diamond-shaped pattern in it !!! This diamond-shaped artifact is a result of the strategy of only updating the 4 pixel neighborhood around each pixel. This is called the ***von Neuman neighborhood***. The diamond-shape can be adjusted to an octagonal shape if a ***Moore neighborhood*** is used. A Moore neighborhood includes the diagonal pixels around a pixel. To use this 8-pixel update model, we would need to add additional code to add border points to the 8 surrounding points:



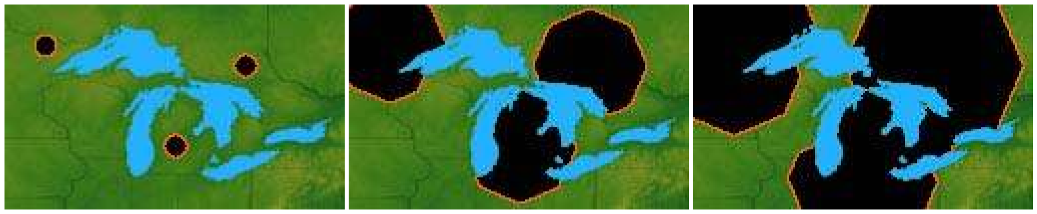


To accomplish this, we simply need to add the following code to the end of our existing code:

```
    if ((x > 0) AND (image[x-1][y-1] is GREEN)) then {
        borders[borderSize] ← new Point (x-1, y-1, distance from (x, y) + 1.417))
        borderSize ← borderSize + 1
        image[x-1][y-1] ← ORANGE
    }
    if ((x < image.width-1) AND (image[x+1][y-1] is GREEN)) then {
        borders[borderSize] ← new Point (x+1, y-1, distance from (x, y) + 1.417))
        borderSize ← borderSize + 1
        image[x+1][y-1] ← ORANGE
    }
    if ((y > 0) AND (image[x-1][y+1] is GREEN)) then {
        borders[borderSize] ← new Point (x-1, y+1, distance from (x, y) + 1.417))
        borderSize ← borderSize + 1
        image[x-1][y+1] ← ORANGE
    }
    if ((y < image.height-1) AND (image[x+1][y+1] is GREEN)) then {
        borders[borderSize] ← new Point (x+1, y+1, distance from (x, y) + 1.417))
        borderSize ← borderSize + 1
        image[x+1][y+1] ← ORANGE
    }

    sort borders by distances from largest to smallest
}
```

Notice that the cost to the diagonals is actually 1.417 ... which is the square root of 2. This is the diagonal distance from the middle of the center pixel to the middle of the diagonal pixel.

As you can see, the result is now octagonal ... a more realistic approximation, although the octagonal shape is still somewhat "artificial-looking":



However, we can make this even more realistic by adding a degree of randomness. Instead of having fixed distances as the fire spreads, we can add some random cost as well. For example, we can adjust the code as follows:

```
10.         borders[borderSize] ← new Point (x-1, y, distance(x, y) + 1 + rand(3))
```

```
14.         borders[borderSize] ← new Point (x+1, y, distance(x, y) + 1 + rand(3))
```

```
18.         borders[borderSize] ← new Point (x, y-1, distance(x, y) + 1 + rand(3))
```

```
22.         borders[borderSize] ← new Point (x, y+1, distance(x, y) + 1 + rand(3))
```

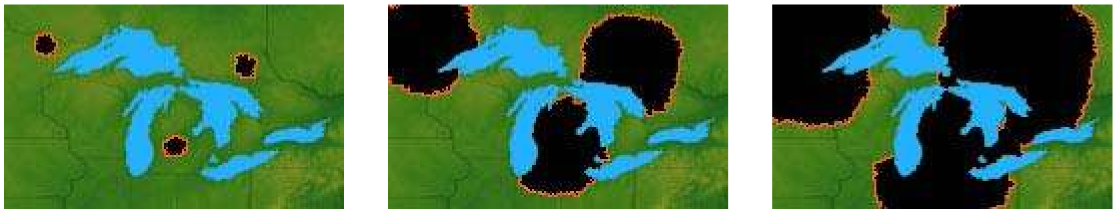
```
26.         borders[borderSize] ← new Point (x-1, y-1, distance(x, y) +1.414+ rand(3))
```

```
30.         borders[borderSize] ← new Point (x+1, y-1, distance(x, y) +1.414 + rand(3))
```

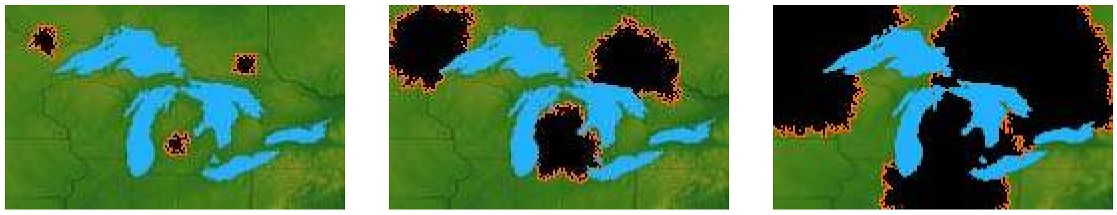
```
34.         borders[borderSize] ← new Point (x-1, y+1, distance(x, y) +1.414 + rand(3))
```

```
38.         borders[borderSize] ← new Point (x+1, y+1, distance(x, y) +1.414 + rand(3))
```

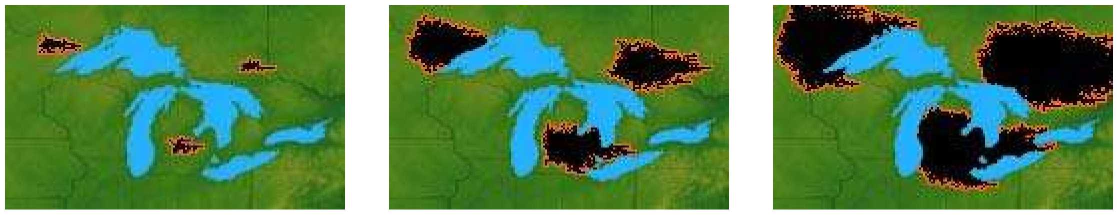
The **rand(3)** indicates a random number from 0 to 3. Here is the result:



The higher the random value, the less circular the shape will be as the fires spread. For example, if we increase the randomness to 50, then here is what we will get:



Lastly, we can even simulate wind. For example, we can have a low cost for spreading to the right and high cost for spreading left ... this would cause the fire to spread quickly rightwards:

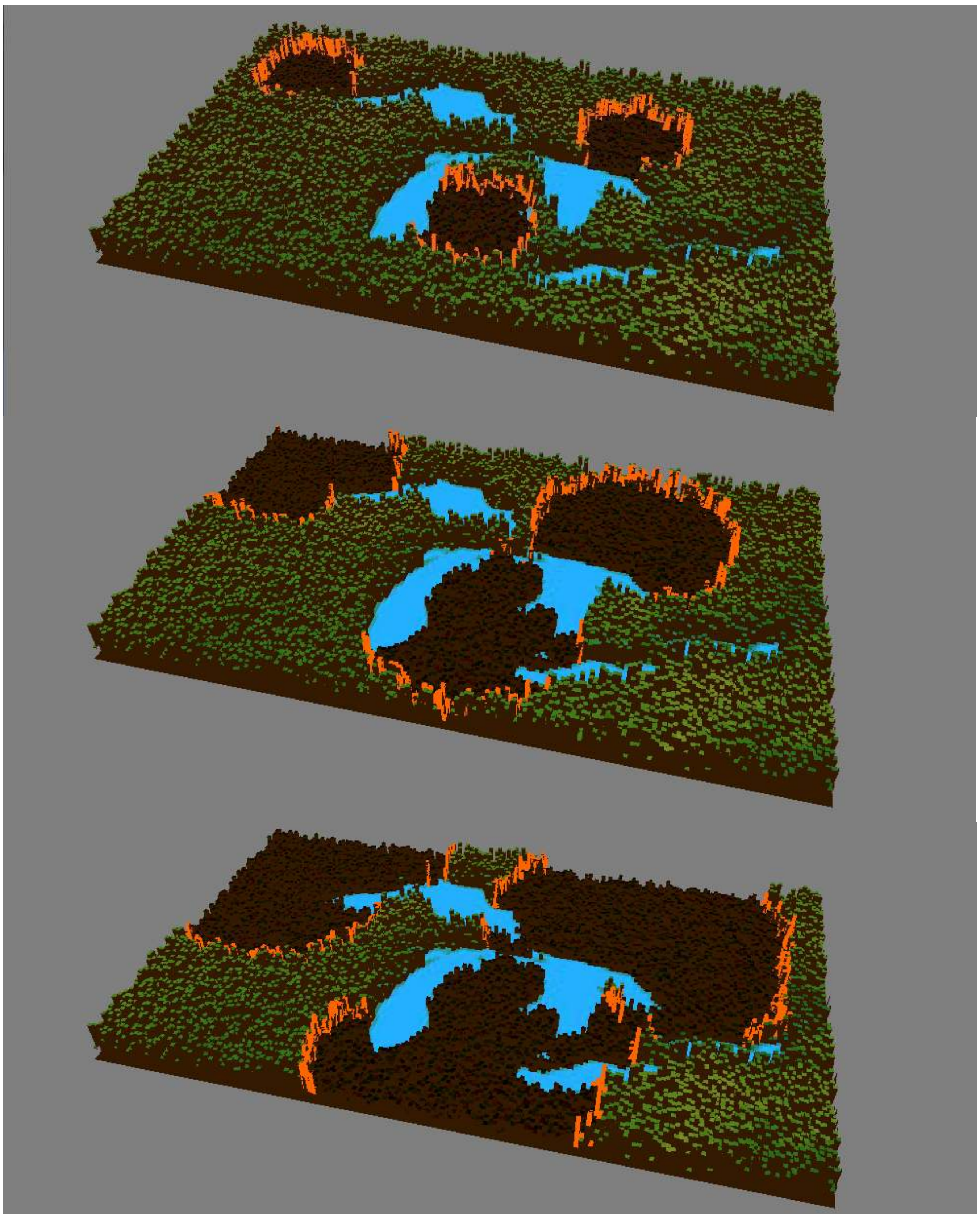


Ultimately, it would be best to produce a more realistic fire-spread model that takes into account the type of trees being burned, wind, elevation, etc...

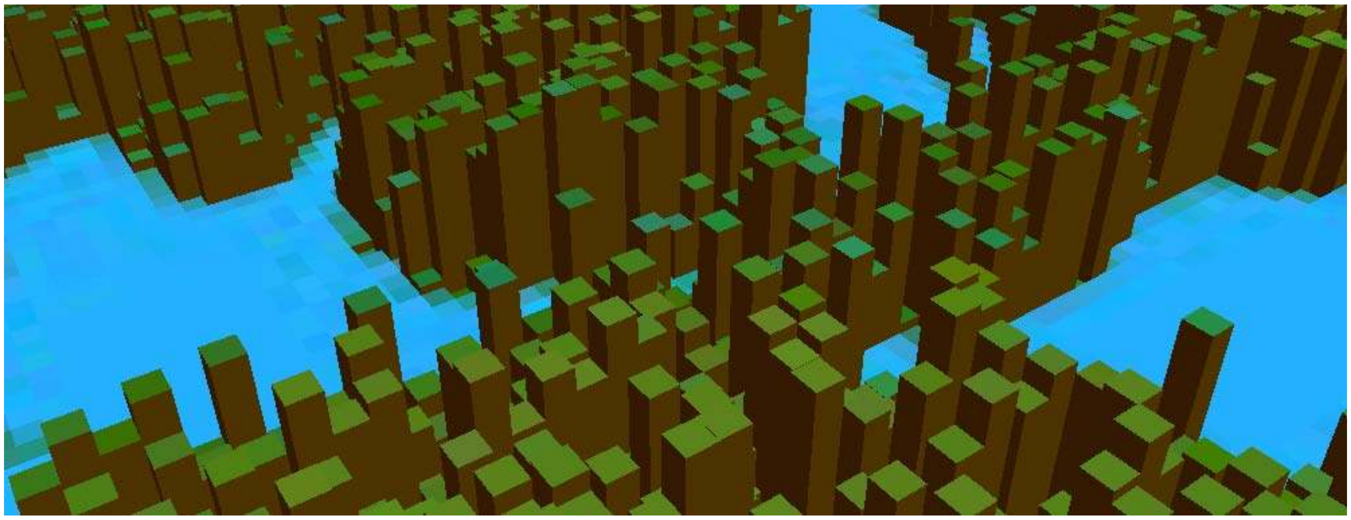
Here is the final code in Processing:

```
PImage     terrain;      // The image with the grass and lakes
Point[]    border;       // points along the border of the fire
int        borderSize;   // # points along the border of the fire

color      orange = color(255, 100, 0);
color      black = color(0, 0, 0);

// Data structure to represent a point
class Point {
  int     x, y;
  float   cost;
  Point(int ix, int iy, float c) {
    x = ix;
    y = iy;
    cost = c;
  }
}

void setup() {
  terrain = loadImage("smallLakes.png");
  size(terrain.width, terrain.height);
  image(terrain, 0, 0);
  loadPixels();
  border = new Point[1000];
  border[0] = new Point(85, 70, 0);
  border[1] = new Point(120, 30, 0);
  border[2] = new Point(20, 20, 0);
  borderSize = 3;
}

void draw() {
  if (!spreadFire()) {
    exit();          // Quit the program
  }
  updatePixels();
  println(borderSize);
}

boolean isUnburnedForest(int x, int y) {
  return (blue(get(x,y)) < 128) && (green(get(x,y)) > 80) && (red(get(x,y)) < 150);
}

boolean spreadFire() {
  if (borderSize > 0) {
    Point p = border[borderSize-1];
    pixels[p.x + (p.y*terrain.width)] = black;
    borderSize--;

    // Check left
    if ((p.x > 0) && isUnburnedForest(p.x-1,p.y)) {
      pixels[p.x-1 + (p.y*terrain.width)] = orange;
      border[borderSize++] = new Point(p.x-1, p.y, p.cost + 1 + random(3));
    }

    // Check right
    if ((p.x < terrain.width-1) && isUnburnedForest(p.x+1,p.y)) {
      pixels[p.x+1 + (p.y*terrain.width)] = orange;
      border[borderSize++] = new Point(p.x+1, p.y, p.cost + 1 + random(3));
    }
    // Check up
    if ((p.y > 0) && isUnburnedForest(p.x,p.y-1)) {
      pixels[p.x + (p.y-1)*terrain.width] = orange;
      border[borderSize++] = new Point(p.x, p.y-1, p.cost + 1 + random(3));
    }

    // Check down
    if ((p.y < terrain.height-1) && isUnburnedForest(p.x,p.y+1)) {
      pixels[p.x + (p.y+1)*terrain.width] = orange;
      border[borderSize++] = new Point(p.x, p.y+1, p.cost + 1 + random(3));
    }

    // Check left/up diagonal
    if ((p.x > 0) && (p.y > 0) && isUnburnedForest(p.x-1,p.y-1)) {
      pixels[p.x-1 + (p.y-1)*terrain.width] = orange;
      border[borderSize++] = new Point(p.x-1, p.y-1, p.cost + 1.414 + random(3));
    }

    // Check right/up diagonal
    if ((p.x < terrain.width-1) && (p.y > 0) && isUnburnedForest(p.x+1,p.y-1)) {
      pixels[p.x+1 + (p.y-1)*terrain.width] = orange;
      border[borderSize++] = new Point(p.x+1, p.y-1, p.cost + 1.414 + random(3));
    }

    // Check left/down diagonal
    if ((p.x > 0) && (p.y < terrain.height-1) && isUnburnedForest(p.x-1,p.y+1)) {
      pixels[p.x-1 + (p.y+1)*terrain.width] = orange;
      border[borderSize++] = new Point(p.x-1, p.y+1, p.cost + 1.414 + random(3));
    }

    // Check right/down diagonal
    if ((p.x < terrain.width-1) && (p.y<terrain.height-1) && isUnburnedForest(p.x+1,p.y+1)) {
      pixels[p.x+1 + (p.y+1)*terrain.width] = orange;
      border[borderSize++] = new Point(p.x+1, p.y+1, p.cost + 1.414 + random(3));
    }

    BubbleSort(border, borderSize);
    return true;
  }
  return false;
}

// BubbleSort
void BubbleSort(Point[] fire, int fireSize) {
  for (int p=fireSize-1; p>=0; p--) {
    boolean madeSwap = false;
    for (int i=0; i<=p-1; i++) {
      if (fire[i].cost < fire[i+1].cost) {
        Point temp = fire[i+1];
        fire[i+1] = fire[i];
        fire[i] = temp;
        madeSwap = true;
      }
    }
    if (!madeSwap) return;
  }
}
```

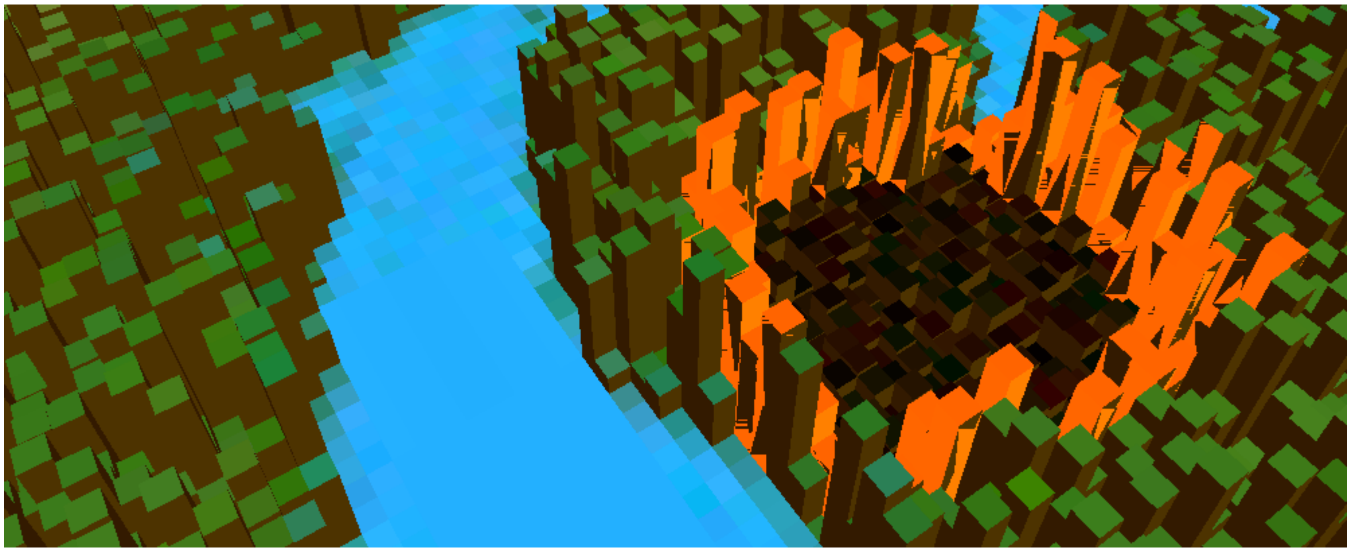
For added enjoyment, we can create a 3D version of the landscape and then watch as the fire spreads, burning the trees (rest assured that no animals were harmed in this simulation). The code to do this is quite similar. Instead of displaying an image, we just need to create a big pile of square surfaces that corresponds to the terrain.



The idea is to simply take the image and assign a height value to each pixel. Water could have height 0 while the trees could have a height which is some constant value with a degree of randomness to provide variety. We could then form a 3D rectangular polyhedra for each pixel and display them ...



As the fire spreads, we can color the rectangular polyhedra orange and then one burned, we can decrease their height to a small value to represent burnt "stubble":



Here is the code. We will not be discussing the 3D aspects of this code:

```
float     xmag, ymag = 0;         // Used for mouse interaction
float     newXmag, newYmag = 0;   // Used for mouse interaction

PImage     terrain;               // The image with the grass and lakes
int[][]    imgPixels;             // Holds the image color data
float[][]  heights;               // Holds the image "pixel" heights
int        halfWidth;             // Half the computerd width
int        halfHeight;            // Half the computerd height
int        scale;                 // Zoom factor
Point[]    border;                // points along the border of the fire
int        borderSize;            // # points along the border of the fire

color      orange = color(255, 100, 0);

// Data structure to represent a point
class Point {
  int     x, y;
  float   cost;
  Point(int ix, int iy, float c) {
    x = ix;
    y = iy;
    cost = c;
  }
}

void setup() {
  size(1000, 1000, P3D);
  noStroke();
  colorMode(RGB, 1);

  terrain = loadImage("smallLakes.png");
  imgPixels = new int[terrain.width][terrain.height];
  heights = new float[terrain.width][terrain.height];
  halfWidth = terrain.width/2;
  halfHeight = terrain.height/2;
  scale = 5;

  for (int i = 0; i < terrain.height; i++) {
    for (int j = 0; j < terrain.width; j++) {
      imgPixels[j][i] = terrain.get(j, i);
      float h = 5*random(2);
      if ((blue(imgPixels[j][i]) > 0.4))
        h = 1;
      heights[j][i] = h;
    }
  }

  border = new Point[1000];
  border[0] = new Point(85, 70, 0);
  border[1] = new Point(120, 30, 0);
  border[2] = new Point(20, 20, 0);
  borderSize = 3;

}

void draw() {
  if (!spreadFire()) {
    println("Total Time: " + millis()/1000.0 + " seconds");
    exit();
  }

  pushMatrix();

  // Code for allowing user interaction to rotate the terrain
  if (mousePressed) {
    if (mouseButton == LEFT) {
      newXmag = mouseX/float(width) * TWO_PI;
      newYmag = mouseY/float(height) * TWO_PI;
    }
    else
      scale = width/2 - mouseX;
  }

  translate(width/2, height/2, -30);
  float diff = xmag-newXmag;
  if (abs(diff) >  0.01) { xmag -= diff/4.0; }
  diff = ymag-newYmag;
  if (abs(diff) >  0.01) { ymag -= diff/4.0; }
  rotateX(-ymag);
  rotateZ(-xmag);
  scale(scale);

  // Draw the terrain
  background(0.5);
  beginShape(QUADS);
  for (int i = 0; i < terrain.width-1; i++) {
    for (int j = 0; j < terrain.height-1; j++) {
      // Set the surface color for this "pixel"
      fill(red(imgPixels[i][j]), green(imgPixels[i][j]), blue(imgPixels[i][j]));

      // Add the surface face
      vertex(i-halfWidth, j-halfHeight, heights[i][j]);
      vertex(i+1-halfWidth, j-halfHeight, heights[i][j]);
      vertex(i+1-halfWidth, j+1-halfHeight, heights[i][j]);
      vertex(i-halfWidth, j+1-halfHeight, heights[i][j]);

      // Add the remaining 4 side faces
      if (imgPixels[i][j] == orange)
        fill(1.0,0.4,0.0);
      else if (heights[i][j] < 10 )
        fill(0.2,0.1,0.0);
      else
        fill(0.2,0.6,0.0);
      vertex(i+1-halfWidth, j-halfHeight, heights[i][j]);
      vertex(i+1-halfWidth, j-halfHeight, heights[i+1][j]);
      vertex(i+1-halfWidth, j+1-halfHeight, heights[i+1][j]);
      vertex(i+1-halfWidth, j+1-halfHeight, heights[i][j]);

      if (i > 0) {
        vertex(i-halfWidth, j-halfHeight, heights[i][j]);
        vertex(i-halfWidth, j+1-halfHeight, heights[i][j]);
        vertex(i-halfWidth, j+1-halfHeight, heights[i-1][j]);
        vertex(i-halfWidth, j-halfHeight, heights[i-1][j]);
      }

      if (imgPixels[i][j] == orange)
        fill(1.0,0.5,0.0);
      else if (heights[i][j] < 10 )
        fill(0.3,0.2,0.0);
      else
        fill(0.3,0.7,0.0);
      vertex(i-halfWidth, j+1-halfHeight, heights[i][j]);
      vertex(i+1-halfWidth, j+1-halfHeight, heights[i][j]);
      vertex(i+1-halfWidth, j+1-halfHeight, heights[i][j+1]);
      vertex(i-halfWidth, j+1-halfHeight, heights[i][j+1]);

      if (j > 0) {
        vertex(i-halfWidth, j-halfHeight, heights[i][j]);
        vertex(i+1-halfWidth, j-halfHeight, heights[i][j]);
        vertex(i+1-halfWidth, j-halfHeight, heights[i][j-1]);
        vertex(i-halfWidth, j-halfHeight, heights[i][j-1]);
      }
    }
  }

  // Draw the bottom face
  fill(0.2,0.1,0);  // Dark brown
  vertex(-halfWidth,  -halfHeight,  -1);  vertex(-halfWidth,  halfHeight,  -1);
  vertex(halfWidth,  halfHeight,  -1);    vertex(halfWidth,  -halfHeight,  -1);

  endShape();
  popMatrix();
}

boolean isUnburnedForest(int x, int y) {
  return (blue(imgPixels[x][y]) < 0.5) && (green(imgPixels[x][y]) > 0.31) &&
(red(imgPixels[x][y]) < 0.59);
}

boolean spreadFire() {
  if (borderSize > 0) {
    Point p = border[borderSize-1];
    imgPixels[p.x][p.y] = color(0.2*random(1), 0.1*random(1), 0);  // random dark brown color
    heights[p.x][p.y] = random(2);      // random height for burned twigs

    borderSize--;

    // Check left
    if ((p.x > 0) && isUnburnedForest(p.x-1,p.y)) {
      imgPixels[p.x-1][p.y] = orange;
      border[borderSize++] = new Point(p.x-1, p.y, p.cost + 1 + random(3));
    }
    // Check right
    if ((p.x < terrain.width-1) && isUnburnedForest(p.x+1,p.y)) {
      imgPixels[p.x+1][p.y] = orange;
      border[borderSize++] = new Point(p.x+1, p.y, p.cost + 1 + random(3));
    }
    // Check up
    if ((p.y > 0) && isUnburnedForest(p.x,p.y-1)) {
      imgPixels[p.x][p.y-1] = orange;
      border[borderSize++] = new Point(p.x, p.y-1, p.cost + 1 + random(3));
    }
    // Check down
    if ((p.y < terrain.height-1) && isUnburnedForest(p.x,p.y+1)) {
      imgPixels[p.x][p.y+1] = orange;
      border[borderSize++] = new Point(p.x, p.y+1, p.cost + 1 + random(3));
    }
    // Check left/up diagonal
    if ((p.x > 0) && (p.y > 0) && isUnburnedForest(p.x-1,p.y-1)) {
      imgPixels[p.x-1][p.y-1] = orange;
      border[borderSize++] = new Point(p.x-1, p.y-1, p.cost + 1.414 + random(3));
    }
    // Check right/up diagonal
    if ((p.x < terrain.width-1) && (p.y > 0) && isUnburnedForest(p.x+1,p.y-1)) {
      imgPixels[p.x+1][p.y-1] = orange;
      border[borderSize++] = new Point(p.x+1, p.y-1, p.cost + 1.414 + random(3));
    }
    // Check left/down diagonal
    if ((p.x > 0) && (p.y < terrain.height-1) && isUnburnedForest(p.x-1,p.y+1)) {
      imgPixels[p.x-1][p.y+1] = orange;
      border[borderSize++] = new Point(p.x-1, p.y+1, p.cost + 1.414 + random(3));
    }
    // Check right/down diagonal
    if ((p.x < terrain.width-1)&&(p.y < terrain.height-1) && isUnburnedForest(p.x+1,p.y+1)) {
      imgPixels[p.x+1][p.y+1] = orange;
      border[borderSize++] = new Point(p.x+1, p.y+1, p.cost + 1.414 + random(3));
    }

    BubbleSort(border, borderSize);
    return true;
  }
  return false;
}


// BubbleSort
void BubbleSort(Point[] fire, int fireSize) {
  for (int p=fireSize-1; p>=0; p--) {
    boolean madeSwap = false;
    for (int i=0; i<=p-1; i++) {
      if (fire[i].cost < fire[i+1].cost) {
        Point temp = fire[i+1];
        fire[i+1] = fire[i];
        fire[i] = temp;
        madeSwap = true;
      }
    }
    if (!madeSwap) return;
  }
}
```

Chapter 7

Recursion

What is in This Chapter ?

This chapter explains the notion of ***recursion*** as it appears in computer science. We discuss briefly the idea of using recursion for simple **math-based problems** and for simple graphical drawing, such as **drawing fractal pictures** which are often found in computer science. Recursion usually involves thinking in a different manner than you may be used to. It is a very powerful technique, however, that can simplify algorithms and make your life easier ... once you understand it. A more thorough discussion of recursive techniques is covered in further courses.

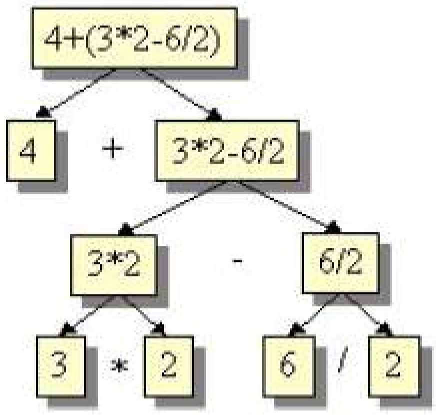
7.1 Recursion

Recall in the first chapter, we discussed the notion of "divide-and-conquer" in regards to breaking a problem down into smaller, more manageable, modular pieces called functions or procedures.

Sometimes when we break a problem down into smaller pieces of the same type of problem, we are simply reducing the amount of information that we must process, since the problem is solved in the same manner ... it is just a smaller version of the same problem.

For example, there are many real-world examples where we often break problems into smaller ones of the same type in order to solve the bigger problem:

1. Jigsaw puzzles are solved in "steps": border, interesting portion, grass, sky, etc..
2. Math problems are broken down into smaller/simpler problems
3. Even climbing stairs eventually breaks down to climbing one step at a time.



Such problems can be solved *recursively*.

> ***Recursion** is a divide-and-conquer technique in which the problem being solved is expressed (or defined) in terms of smaller problems of the same type.*

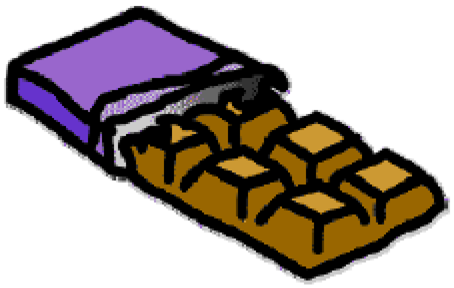
The word ***Recursion*** actually comes from a Latin word meaning "a running back". This makes sense because recursion is the process of actually "going off" and breaking down a problem into small pieces and then bringing the solutions to those smaller pieces back together to form the complete solution. So ...

- recursion **breaks down** a complex problem into **smaller sub-problems**
- the sub-problems are **smaller** instances of the **same type** of problem.

So, when using recursion, we need to consider how to:

1. break the problem down into smaller sub-problems
2. deal with each smaller sub-problem
3. merge the results of the smaller sub-problems to answer the original problem

In fact, it is actually easier than this sounds. Most of the time, we simply take the original problem and **break/bite off** a small piece that we can work with. We simply keep biting off the small pieces of the problem, solve them, and then merge the results.



Consider the simple example of computing the factorial of an integer. You may recall that the function works as follows:

```
0! = 1
1! = 1
2! = 2x1
3! = 3x2x1
4! = 4x3x2x1
5! = 5x4x3x2x1
etc..
```

So, the function is defined non-recursively as follows:

```
1                                         if N = 0
N! = N x (N-1) x (N-2) x ... x 3 x 2 x 1   if N ≥ 1
```

If you were asked to write a factorial function, you could do so with a FOR loop as follows:

```
Function: Factorial
    n:      a positive integer for which to find the factorial

1.  answer ← 1
2.  for each integer i from 2 to n do {
3.      answer ← answer * i
    }
4.  return answer
```

This code is simple and straight-forward to write.

The factorial problem, however, can also be defined recursively as follows:

```
1                    if N = 0
N! = N x (N-1)!      if N ≥ 1
```

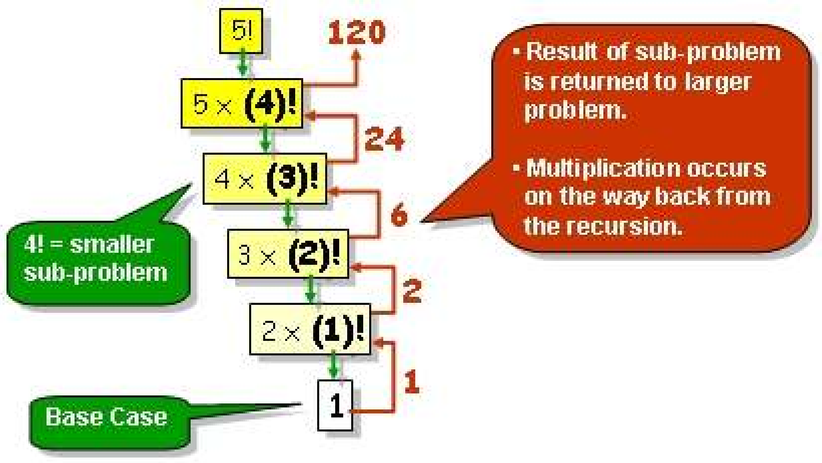
Notice how **N!** is expressed as a function of the smaller sub-problem of **(N-1)!** which is of the same type of problem (i.e., it is also a factorial problem, but for a smaller number).

So, if we had the solution for **(N-1)!**, say **S**, we could use this to compute the solution to **N!** as follows: `N! = N x S`

However, how do we get the solution for **(N-1)!** ? We use the same formula, setting **N** to **N-1** as follows:

```
(N-1)! = (N-1) x ((N-1)-1)!   =   (N-1) x (N-2)!
```

Similarly, we can recursively break down **(N-2)!** in the same way. Eventually, as we keep reducing **N** by **1** each time, we end up with **N=0 or 1** and for that simple problem we know the answer is **1**. So breaking it all down we see the solution of **5!** as follows:

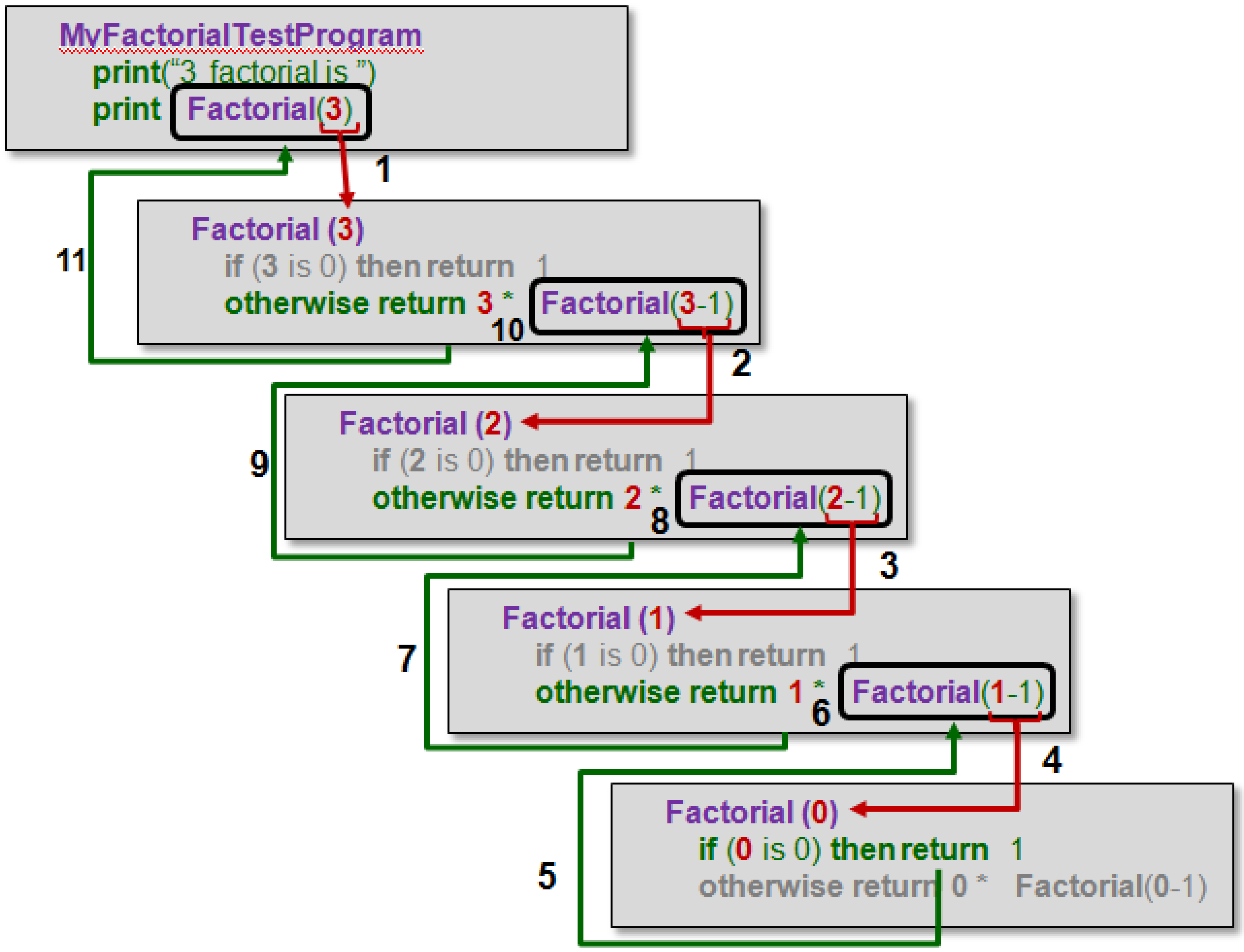

5!
5 × (4)!
4 × (3)!
3 × (2)!
2 × (1)!
1
120
24
6
2
1
4! = smaller sub-problem
Base Case
• Result of sub-problem is returned to larger problem.
• Multiplication occurs on the way back from the recursion.

If you were asked to write this **recursive** factorial function, you could do so as follows:

```
Function: FactorialRecursive
        n:          a positive integer for which to find the factorial

1.      if (n is 0) then
2.              answer ← 1
3.      otherwise
4.              answer ← n * FactorialRecursive(n-1)
5.      return answer
```

Notice how we no longer have a repeating loop in our code. Instead, the **FactorialRecursive** function calls itself with a smaller value of **n**. That is, the function repeatedly calls itself over and over again with smaller values of **n** until finally **n** has the smallest value of **0**, in which case the code stops calling itself recursively. A recursive function is easily identifiable by the fact that a function or procedure calls itself. The "stopping situation" is called the ***base case*** of the problem. In general, a recursive function (or procedure) may have multiple base cases (i.e., multiple stopping conditions).

To understand the inner-workings of the function as it calls itself, consider the following diagram which shows how the control passes from one function call to the next when computing the factorial of 3. There are 11 main "steps" of the recursion as shown numbered in black bold font:


MyFactorialTestProgram
print("3 factorial is ")
print Factorial(3)
1
11
Factorial (3)
if (3 is 0) then return 1
otherwise return 3 * Factorial(3-1)
10
2
9
Factorial (2)
if (2 is 0) then return 1
otherwise return 2 * Factorial(2-1)
8
3
7
Factorial (1)
if (1 is 0) then return 1
otherwise return 1 * Factorial(1-1)
6
4
5
Factorial (0)
if (0 is 0) then return 1
otherwise return 0 * Factorial(0-1)

If you were to compare the non-recursive solution with the recursive solution ... which do you find to be simpler ?

<u>Function: Factorial</u>
n: a positive integer
 	
1. answer ← 1
2. for each integer i from 2 to n do {
3.   answer ← answer * i
    }
4. return answer
	<u>Function: Factorial</u>
	n: a positive integer
	
	1. if (n is 0) then
	2. answer ← 1
	3. otherwise
	4. answer ← n * Factorial(n-1)
	5. return answer

You may feel that the non-recursive version is simpler. In this particular example, you are probably right. So the question that arises is ... "Why should we use recursion ?".

Here are 4 reasons:

1. Some problems are **naturally** recursive and are easier to express recursively than non-recursively. For example, some math problems are defined recursively.

2. Sometimes, using recursion results in a simpler, **more elegant solution**.

3. Recursive solutions may actually be **easier to understand**.

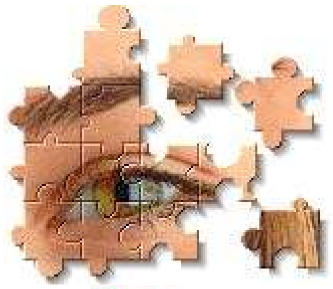
4. In some cases, a recursive solution may be the **only way to approach** a seemingly **overwhelming problem**.

As we do various examples, the advantages of using recursion should become clear to you.

How do we write our own recursive functions and procedures ? It is important to remember the following two very important facts about the sub-problems:

- must be an instance of the **same kind** of problem
- must be **smaller** than the original problem

The trick is to understand how to "bite off" small pieces of the problem and knowing when to stop doing so. When we write the code, we will usually start with the base cases since they are the ones that we know how to handle. For example, if we think of our "real world" examples mentioned earlier, here are the base cases:



1. For the **jigsaw puzzle**, we divide up the pieces until we have just a few (maybe 5 to 10) pieces that form an interesting part of our picture. We stop dividing at that time and simply solve (by putting together) the simple base case problem of this small sub-picture. So the base case is the problem in which there are only a few pieces all part of some identifiable portion of the puzzle.

2. For the **math problem**, we simply keep breaking down the problem until either (a) there is a simple expression remaining (e.g., 2 + 3) or (b) we have a single number with no operations (e.g., 7). These are simple base cases in which there is either one operation to do or none.

3. For the **stair climbing problem**, each stair that we climb brings us closer to solving the problem of climbing all the stairs. Each time, we take a step up, the problem is smaller, and of the same type. The base case is our simplest case when there is only one stair to climb. We simply climb that stair and there is no recursion to do at that point.

So then to write recursive methods, we first need to understand "where" the recursion will fit in. That is, we need to understand how to reduce the problem and then merge the results. It is important to fully understand the problem and the kind of parameters that you will need as the function/procedure gets called repeatedly. Make sure you understand how the problem gets smaller (i.e., usually the parameters change in some way). Then, you can implement the simple base cases and add in the recursion. Let us look at a few examples.

7.2 Math Examples

Recursive math examples are easy to program because we are often given the recursive definition directly and then it easily transfers into a program.

Example:

For example, consider computing how much money per month a person would have to pay to pay back a mortgage on his/her home. Consider the following notation:

- **amount** = mortgage amount borrowed (e.g., $130,000)
- **rate** = annual interest rate as a percentage (e.g., 3.5%)
- **months** = length of time for the mortgage to be paid off (e.g., 300 months for a 25 year mortgage)
- **payment(amount, rate, months)** = monthly payment to pay off mortgage (e.g., $647.57)

We would like to calculate the value for the **payment** function. Here is the function and its parameters as defined:

Function: Payment	
amount:	the principal amount of the mortgage
rate:	annual interest rate to be paid
months:	mortgage term (in months)

If we borrow money for 0 months, then the whole point of borrowing is silly. Assume then that **months** > 0. What if **months** = 1 ... that is ... we want to pay off the entire mortgage one month after we borrowed the money ? Then we must pay one month interest and so the value of the **payment** function should return:

amount + **amount***(**rate**/12) or **amount** * (1 + (**rate**/12))

which is one month of interest on the original loan amount. We divide by 12 since the rate is per year and we want only 1 month of interest. For a 2-month term, the payment would be:

[amount * (1 + (**rate**/12))**]** * (1 + (**rate**/12))

and for a 3-month term:

{[amount * (1 + (**rate**/12))**]** * (1 + (**rate**/12))**}** * (1 + (**rate**/12))

So you can see, the interest to be paid each month is compounded monthly...so we pay interest on interest owed and this grows larger each month.

There are a few ways to compute mortgage payments, but we will use the following formula so that we can practice our recursion. The formula for computing the payment is actually recursive and is defined as follows (showing first letter only for each parameter name):

$$p(a, r, m) = \frac{a}{\left\{ \dfrac{a}{p(a, r, m-1)} + \dfrac{1}{(1 + r/12)^{m}} \right\}}$$

Looking at the formula, do you see that the problem is recursive ? The result of the payment function **p(a,r,m)** depends on the result of the payment function **p(a,r,m-1)** for one less month. It is easy to see that the problem for one less month is the same problem (i.e., same function) and that the problem is smaller.

Now let us determine the base case (i.e., the simplest case). Well, the **amount** and **rate** do not change throughout the computation. Only the **month** value changes ... by getting smaller. So the base case must be the stopping condition for the months being decreased. This base case is the simplest case which follows from the definition.

```
Function: Payment
        amount:     the principal amount of the mortgage
        rate:       annual interest rate to be paid
        months:     mortgage term (in months), assumed to be > 0

1.      if (months is 1) then
2.          return amount * (1 + rate/12)
```

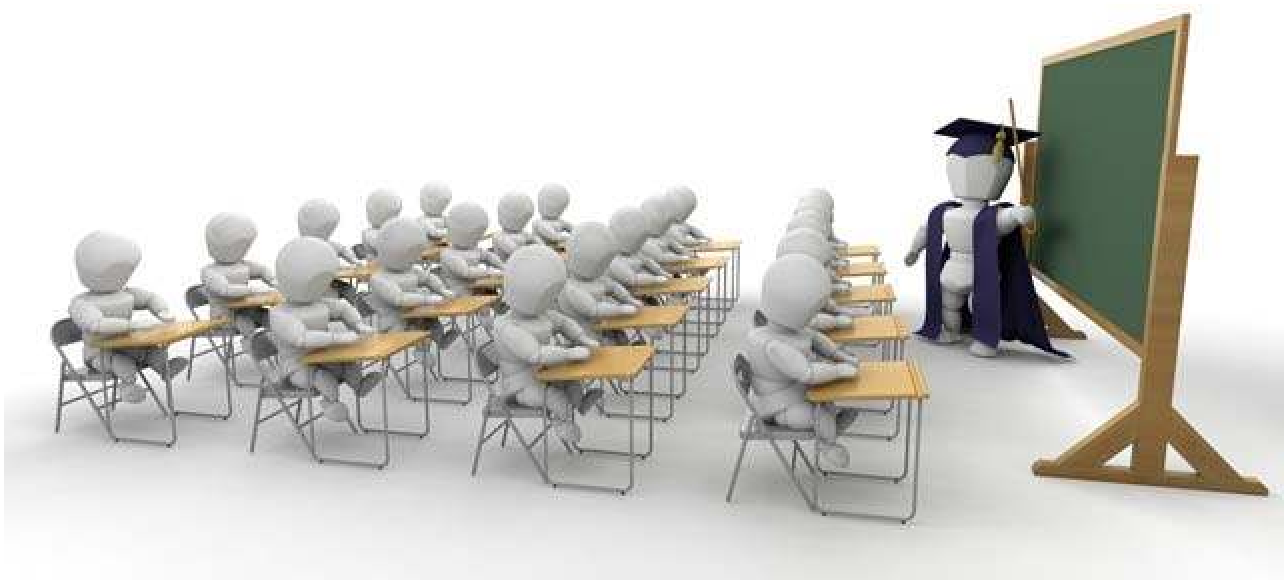
That wasn't so bad. Now what about the recursive part ? It also follows from the formula:

```
Function: Payment
        amount:     the principal amount of the mortgage
        rate:       annual interest rate to be paid
        months:     mortgage term (in months), assumed to be > 0

1.      if (months is 1) then
2.          return amount * (1 + rate/12)
3.      return amount / ((amount / Payment(amount, rate, months-1)) + (1 + rate/12)^months)
```

As you can see, the code is easy to write as long as we have a recursive definition to begin with.

Example:

Imagine now that we have a teacher who wants to choose 10 students from a class of 25, to participate in a special project. Assuming that each student is unique in his/her own way ... how many different groups of students can be chosen ?



This is a classic problem using binomial coefficients that is often encountered in combinatorics for determining the amount of variety in a solution, giving insight as to the complexity of a problem. In general, if we have **n** items and we want to choose **k** items, we are looking for the solution of how many groups of **k**-elements can be chosen from the **n** elements.

The expression is often written as follows and pronounced "n choose k": $\binom{n}{k}$

Assuming that **n** is fixed, we can vary **k** to obtain a different answer. The simplest solutions for this problem are when **k**=0 or **k**=**n**. If **k**=0, we are asking how many groups of zero can be chosen from **n** items. The answer is **1** ... there is only one way to choose no items. Similarly, if **k**=**n** then we want to choose all items ... and there is only one way to do that as well. Also, if **k**>**n**, then there is no way to do this and the answer is zero (e.g., cannot choose 10 students from a group of 6).

Otherwise, we can express the problem as a recursive solution by examining what happens when we select a single item from the **n** items. Imagine therefore that we select one particular student that we definitely want to participate in the project (e.g., the teacher's "pet").

We need to then express the solution to the problem, taking into account that we are going to use that one student. In the remainder of the classroom, we therefore have **n**-1 students left and we still need to select **k**-1 students. Here is what is left → $\binom{n-1}{k-1}$

The other situation is that we decide that we definitely DO NOT want a particular student to participate in the project (e.g., he's been a "bad boy" lately). In this case, if we eliminate that student, we still need to select **k** students, from the **n**-1 remaining. Here is what is left → $\binom{n-1}{k}$

Now, I'm sure you will agree that if we examine one particular student ... then we will either select that student for the project or not. These two cases represent all possible solutions to the problem. Therefore, we can express the entire solution as:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

This is a recursive definition. The problem is of the same type and gets smaller each time (since **n** gets smaller and **k** does also in one case). This problem is interesting because there are two recursive calls to the same problem. That is, the problem branches off twice recursively.

So then, to write the code, we start with the function:

```
Function: StudentCombinations
        n:  total number of students
        k:  number of students to choose
```

What about the base case(s) ? Well, we discussed the situations where **k**=**n**, **k**=0, **k**>**n**, so these are the simplest cases:

```
Function: StudentCombinations
        n:  total number of students
        k:  number of students to choose

1.      if (k > n) then
2.          return 0
3.      if (k is 0) then
4.          return 1
5.      if (k is n) then
6.          return 1
```

As before, the recursive case is easy, since we have the definition. This time, the code will be interesting as the function will call itself twice:

```
Function: StudentCombinations
        n:  total number of students
        k:  number of students to choose

1.      if (k > n) then
2.          return 0
3.      if (k is 0) then
4.          return 1
5.      if (k is n) then
6.          return 1

7.      return StudentCombinations(n-1,k-1) + StudentCombinations(n-1,k)
```

Not too bad ... was it ? Do you understand how the recursion will eventually stop ?

7.3 Graphical Examples

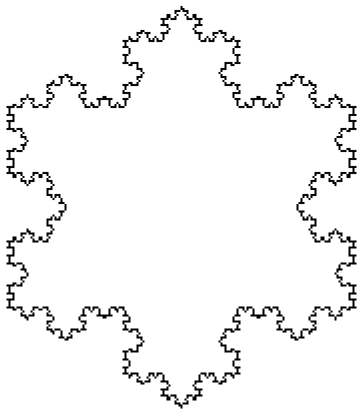
Recursion also appears in many graphical applications. For example, in computer games, trees are often created and displayed using recursion. Fractals are often used in computer graphics.

A ***fractal*** *is a rough or fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole. (wikipedia)*

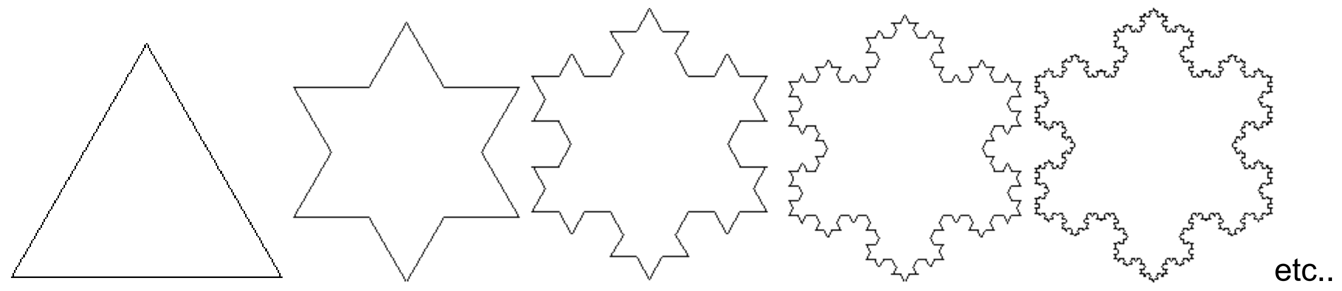
So, a fractal is often drawn by repeatedly re-drawing a specific pattern in a way that the pattern becomes smaller and smaller, but is of the same type.

Example:

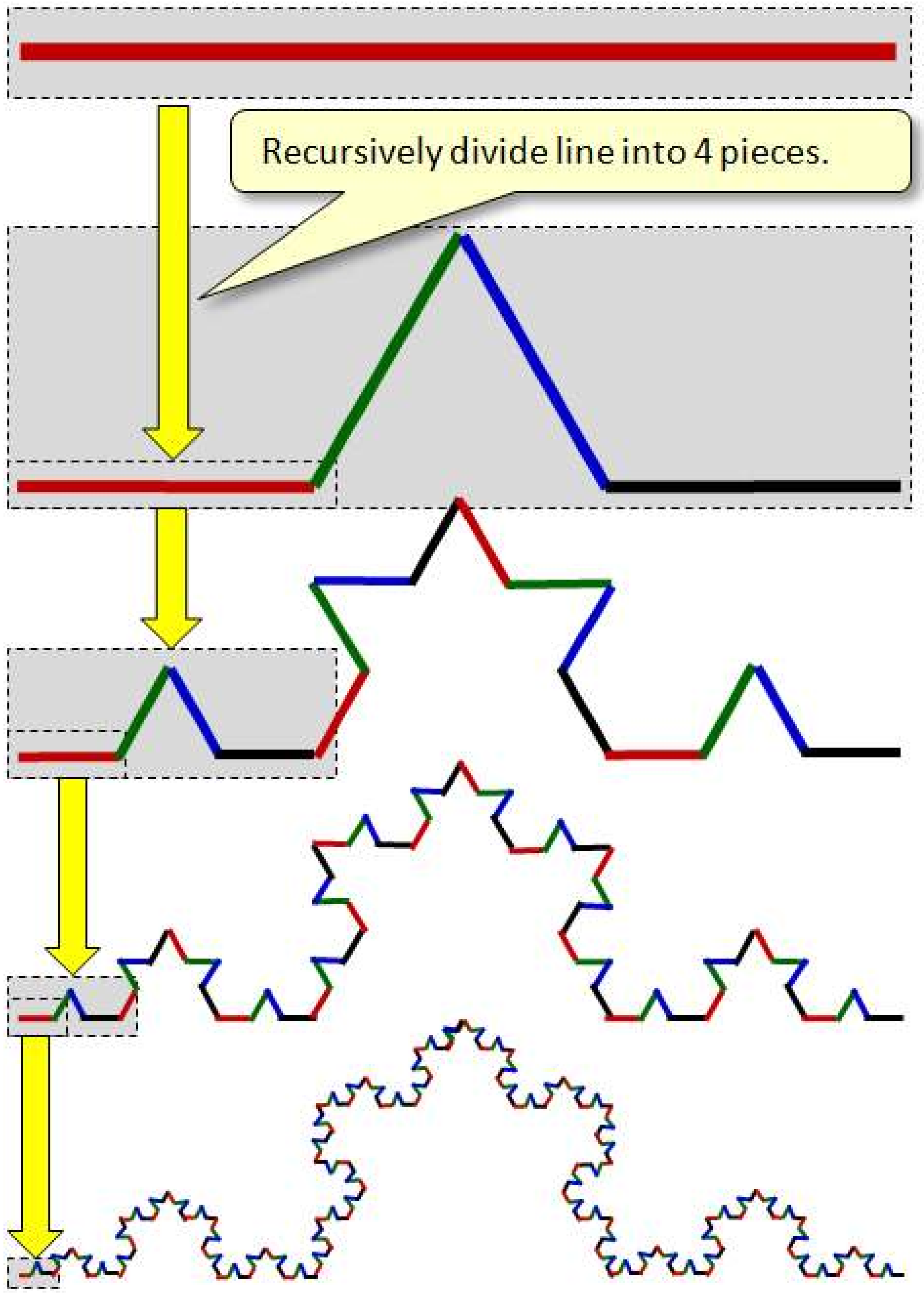
Consider how we can draw a snowflake. See if you can detect the recursion in the following snowflake (called the **Koch Snowflake**):



It may not be easy to see. However ... if we were to examine the process of creating this snowflake recursively, notice how it can be done by starting with an equilateral triangle:

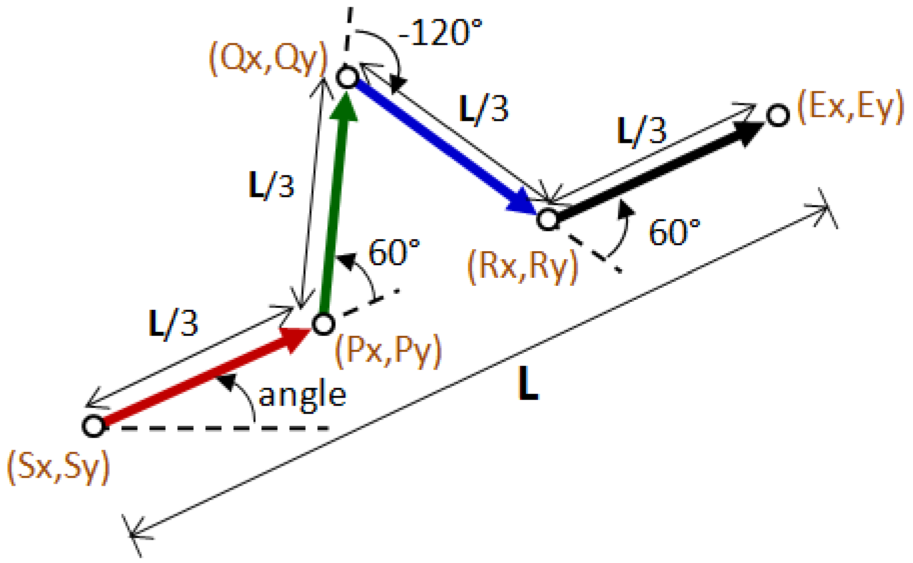


The recursion happens on each side of the triangle. Consider a single horizontal edge. We can recursively alter the edge into 4 segments as follows:



To accomplish this, we need to understand how a single edge is broken down into 4 pieces. Then we need to understand how this gets done recursively. Notice above how each recursive edge gets smaller. In fact, during each step of the recursion, the edge gets smaller by a factor of 3. Eventually, the edge will be very small in length (e.g., 2) and the recursion can stop, since breaking it down any further will not likely make a difference graphically.

Consider an edge going from a starting location **(Sx, Sy)** and going in direction **angle** towards location **(Ex, Ey)**. Here is how we break the edge down into 4 pieces:



So the code for doing this, is as follows:

The additional value of **180°** in the **cosine** function is used because of the reverse-**y**-coordinate system that is used in computer graphics. In mathematics, the origin **(0,0)** of the coordinate system is in the **bottom** left corner. In computer graphics, however, the origin **(0,0)** is at the **top** left corner. By adding **180°** to our angles we are compensating for the different origin, essentially flipping the **y**-coordinate.

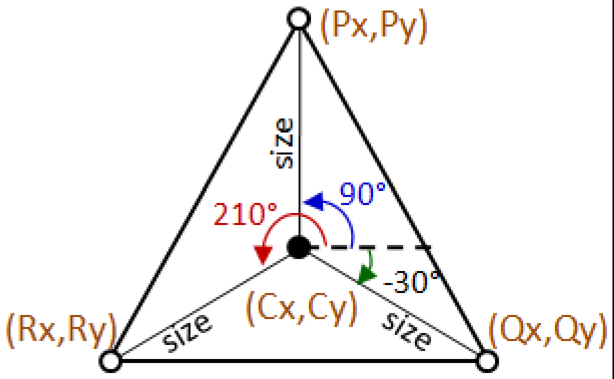
```
Function: drawKochEdge
        Sx:      x location to start drawing at
        Sy:      y location to start drawing at
        Ex:      x location to stop drawing at
        Ey:      y location to stop drawing at
        angle:   direction to draw towards

1.      length ← distance from (Sx, Sy) to (Ex, Ey)/3
2.      Px ← Sx + length * cos(angle)
3.      Py ← Sy + length * sin(angle+180°)
4.      Qx ← Px + length * cos(angle + 60°)
5.      Qy ← Py + length * sin(angle + 60°+180°)
6.      Rx ← Qx + length * cos(angle - 60°)
7.      Ry ← Qy + length * sin(angle - 60°+180°)
8.      draw line from (Sx, Sy) to (Px, Py)
9.      draw line from (Px, Py) to (Qx, Qy)
10.     draw line from (Qx, Qy) to (Rx, Ry)
11.     draw line from (Rx, Ry) to (Ex, Ey)
```

The above code will work for any start/end locations. The **angle** is also passed in as a parameter, but alternatively it can be computed from the start and end points of the edge. It is important that the angle passed is actually the correct angle from the start to the end location.

Now, how do we draw the entire snowflake ? We need to draw three edges to represent the starting triangle. Assume that we want the snowflake centered at some location **(Cx, Cy)** and that the snowflake's size is defined by the length of the three triangle edges called **size**.

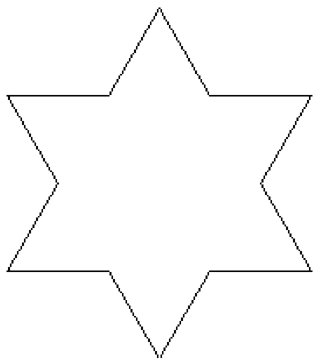
We can draw the triangle with simple straight line edges as follows:

```
Function: drawSnowflake
        Cx, Cy:   location of center of snowflake
        size:     size of snowflake (i.e., length of inner triangle edge)

1.      Px ← Cx + size * cos(90°)
2.      Py ← Cy + size * sin(90°+180°)
3.      Qx ← Cx + size * cos(-30°)
4.      Qy ← Cy + size * sin(-30°+180°)
5.      Rx ← Cx + size * cos(210°)
6.      Ry ← Cy + size * sin(210°+180°)
7.      draw line from (Px, Py) to (Qx, Qy)
8.      draw line from (Qx, Qy) to (Rx, Ry)
9.      draw line from (Rx, Ry) to (Px, Py)
```



We can draw the triangle with a single Koch edge as follows:

```
Function: drawSnowflake
        Cx, Cy:   location of center of snowflake
        size:     size of snowflake (i.e., length of inner triangle edge)

1.      Px ← Cx + size * cos(90°)
2.      Py ← Cy + size * sin(90°+180°)
3.      Qx ← Cx + size * cos(-30°)
4.      Qy ← Cy + size * sin(-30°+180°)
5.      Rx ← Cx + size * cos(210°)
6.      Ry ← Cy + size * sin(210°+180°)
7.      drawKochEdge(Px, Py, Qx, Qy, -60°)
8.      drawKochEdge (Qx, Qy, Rx, Ry, 180°)
9.      drawKochEdge (Rx, Ry, Px, Py, 60°)
```



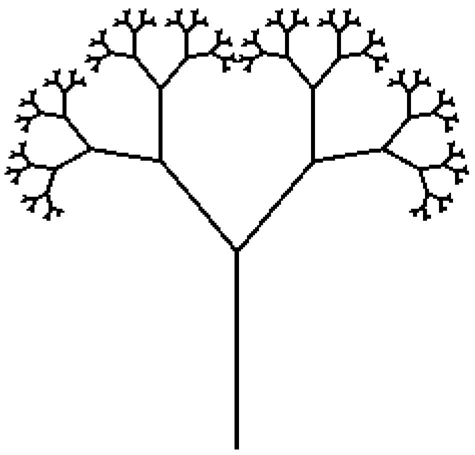
The code above will draw the shape with one level of recursion. But how do we complete the code so that it does many levels of recursion ? We need to come up with a base case for stopping the recursion. This can be done in the **drawKochEdge** procedure by noticing when the length of the edge is very small (e.g., 2). In this case, we draw a regular line as opposed to breaking it up into 4 and recursively drawing. Notice the code:

```
Function: drawKochEdge
        Sx, Sy:      location to start drawing at
        Ex, Ey:      location to stop drawing at
        angle:       direction to draw towards

1.      length ← distance from (Sx, Sy) to (Ex, Ey) / 3
2.      if (length < 2) then
3.          draw line from (Sx, Sy) to (Ex, Ey)          // just draw a straight line
        otherwise {
4.          Px ← Sx + length * cos(angle)
5.          Py ← Sy + length * sin(angle+180°)
6.          Qx ← Px + length * cos(angle + 60°)
7.          Qy ← Py + length * sin(angle + 60°+180°)
8.          Rx ← Qx + length * cos(angle - 60°)
9.          Ry ← Qy + length * sin(angle - 60°+180°)
10.         drawKochEdge(Sx, Sy, Px, Py, angle)
11.         drawKochEdge(Px, Py, Qx, Qy, angle + 60°)
12.         drawKochEdge(Qx, Qy, Rx, Ry, angle - 60°)
13.         drawKochEdge(Rx, Ry, Ex, Ey, angle)
        }
```

This code will produce the desired snowflake pattern.

Example:

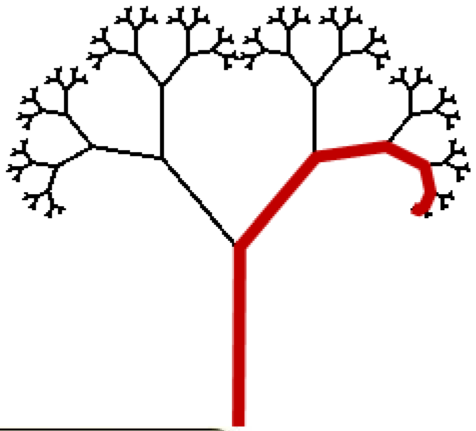
Consider drawing a tree. We can draw the stem of a tree as a branch and then repeatedly draw smaller and smaller branches of the tree until we reach the end of the tree. Here, for example, is a fractal tree →



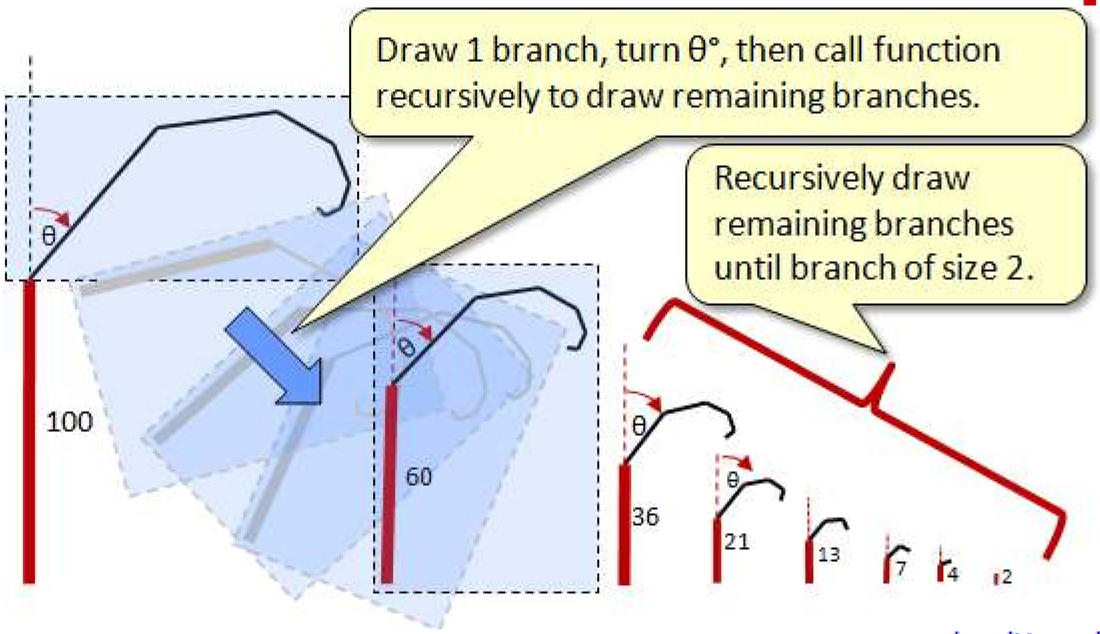
How can we draw this ? Well, imagine drawing just one portion of this tree starting at the bottom facing upwards. We can travel for some number of pixels (e.g., 100) and then bend to the right, perhaps 40° as shown here. Then, we can draw for a smaller distance, perhaps 60% of the previous branch. We can do this repeatedly, turning 40° and drawing a new smaller branch. We can stop when the branch is small enough (perhaps 2 or 3 pixels in length).

Consider doing this for just one portion of branches in the tree (e.g, the right sequence of branches shown here in bold/red).

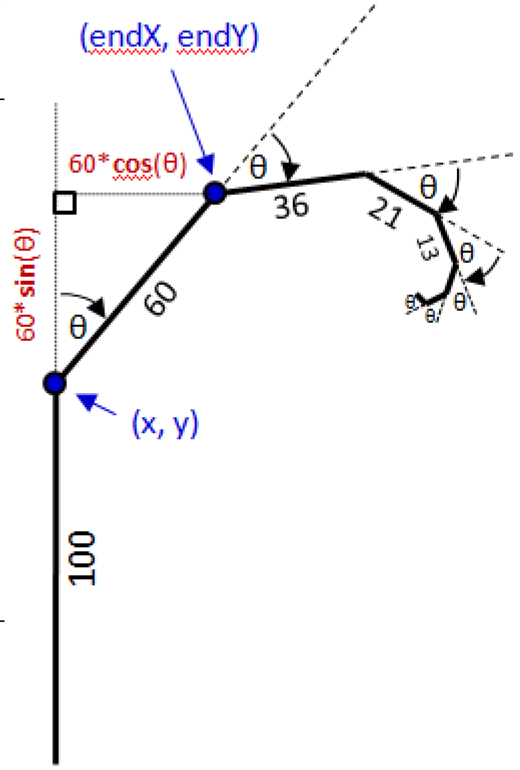
Assume that the **bendAngle** is set at 40° and that the **shrinkFactor** is 0.6 (i.e., 60%). Consider the code to draw the rightmost set of branches of the tree. Assume that the initial **(x,y)** location is centered at the bottom of the window and that the drawing direction is north (i.e., upwards at 90°).



Each time we can draw a single branch, then turn **bendAngle** and then recursively draw the remaining branches starting with branch length reduced by the **shrinkFactor**.


Draw 1 branch, turn θ°, then call function recursively to draw remaining branches.
Recursively draw remaining branches until branch of size 2.
θ
100
60
36
21
13
7
4
2

Here is the code:

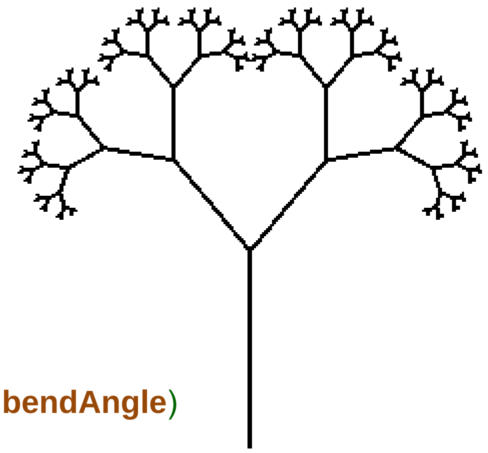


```
Function: drawSingleTreePath
        length:     length of tree branch
        x, y:       location to start drawing at
        dir:        direction to draw towards

        if (length < 2)
            return
        endX ← x + length * cos(dir + bendAngle)
        endY ← y + length * sin(dir + bendAngle+180°)
        draw line from (x, y) to (endX, endY)
        drawTree(length * shrinkFactor, endX,
                 endY, dir + bendAngle)
```

Assume that we begin the tree by drawing a 100 pixel length vertical line from the bottom center of the screen (**windowWidth**/2, **windowHeight**) facing upwards (i.e. North at 90°) to (**windowWidth** /2, **windowHeight** -100) We would complete the drawing simply by calling the function with initial values corresponding to facing North and a 100***shrinkFactor** = 60 pixel length as follows:

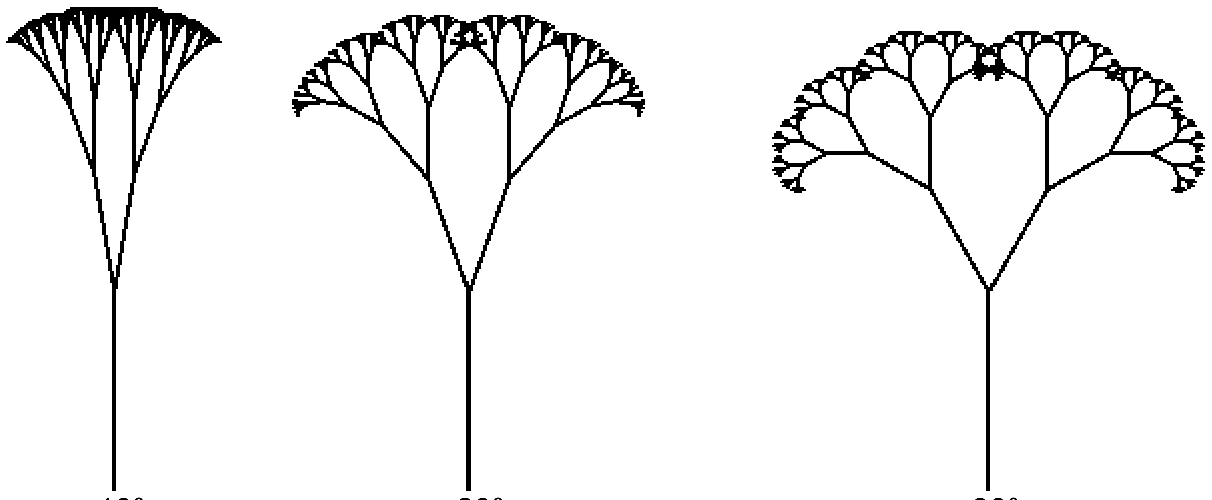
drawSingleTreePath (60, **windowWidth** /2, **windowHeight** -100, 90°)

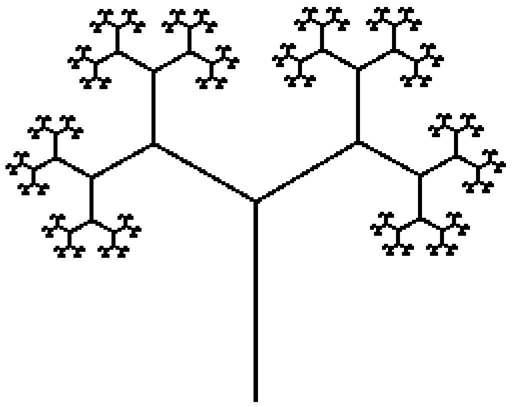
The recursion stops when the tree branch size reaches 2, which is somewhat arbitrary. To draw the remainder of the tree, we simply add code to draw left as well:

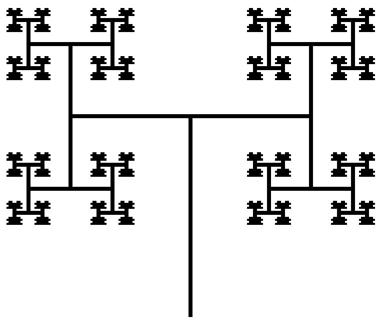


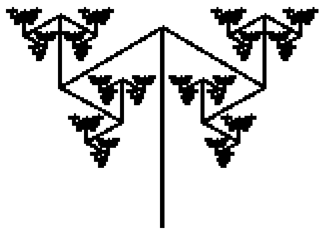
```
Function: drawTree
        length:     length of tree branch
        x, y:       location to start drawing at
        dir:        direction to draw towards

1.      if (length < 2)
2.          return
3.      endX ← x + length * cos(dir + bendAngle)
4.      endY ← y + length * sin(dir + bendAngle+180°)
5.      draw line from (x, y) to (endX, endY)
6.      drawTree(length * shrinkFactor, endX, endY, dir + bendAngle)

7.      endX ← x + length * cos(dir - bendAngle)
8.      endY ← y + length * sin(dir - bendAngle+180°)
9.      draw line from (x, y) to (endX, endY)
10.     drawTree(length * shrinkFactor, endX, endY, dir - bendAngle)
```

Notice how the tree takes shape because each time we draw a single branch, we branch out 40° left and 40° right. Notice how the trees differ as we change the **bendAngle** to other angles other than 40°:

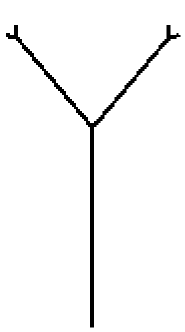


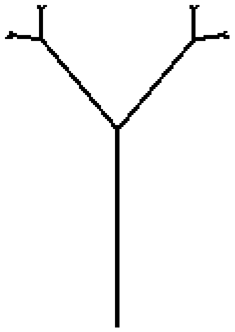

60°

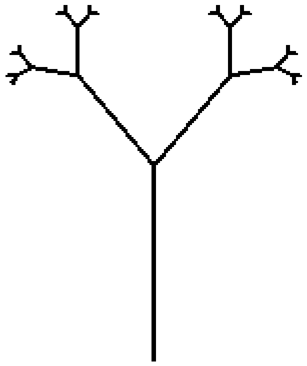

90°

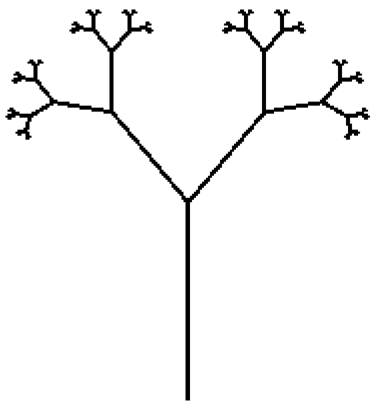

120°

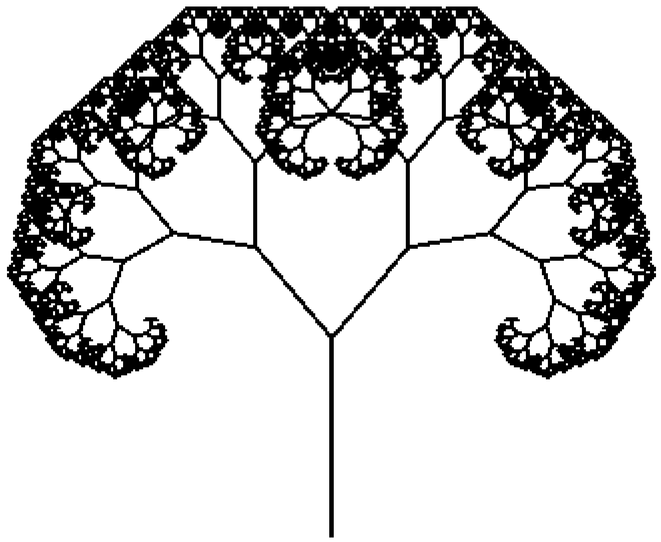
Also, as we adjust the **shrinkFactor** to other values, notice how the tree changes (here the **bendAngle** is fixed at 45°:

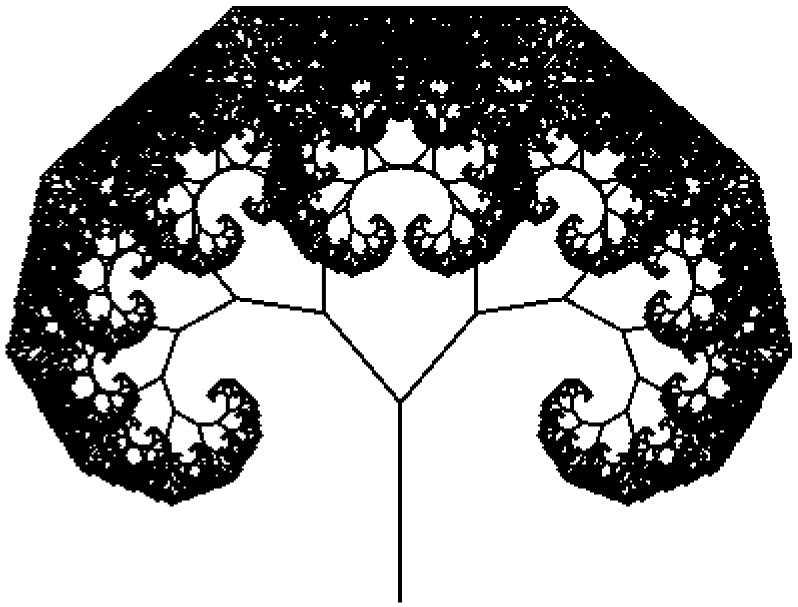

0.10


0.25


0.4


0.5


0.7


0.75

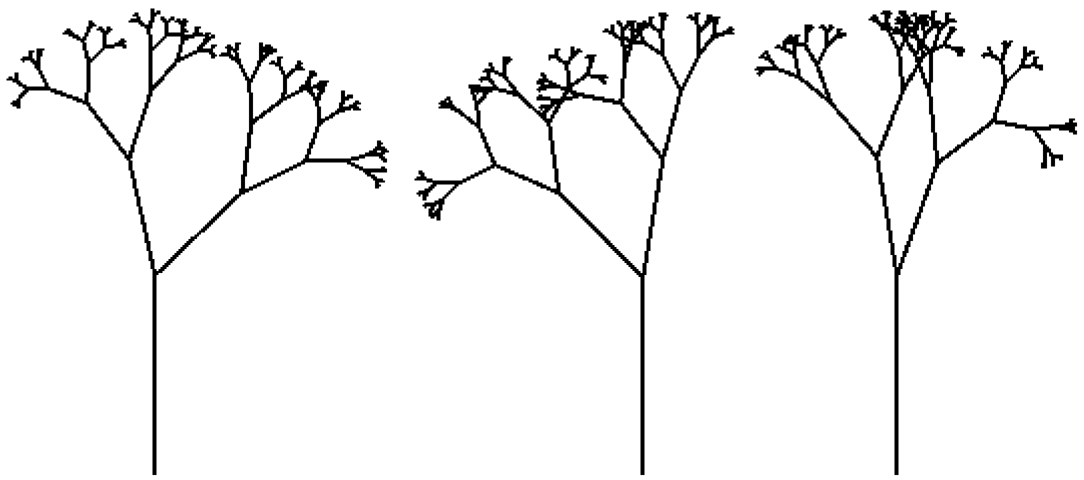
Example:

Now, consider drawing a more realistic tree. A real tree does not have branches that grow at exact angles. Let us begin by adjusting the **bendAngle** to be random instead of fixed at say **40°**. That way, each branch of the tree can bend randomly. Of course, we want to have some limits as to how random they should bend.

We can change the **bendAngle** so that it bend at least some amount each time (e.g., π/32 = 5.6°) plus some random amount (e.g., 0 to π/4 ... or 0° to 45°):

```
bendAngle ← 5.6° + (random value from 0° to 45°)
```

The result is that our trees will look less symmetrical and will bend more naturally:



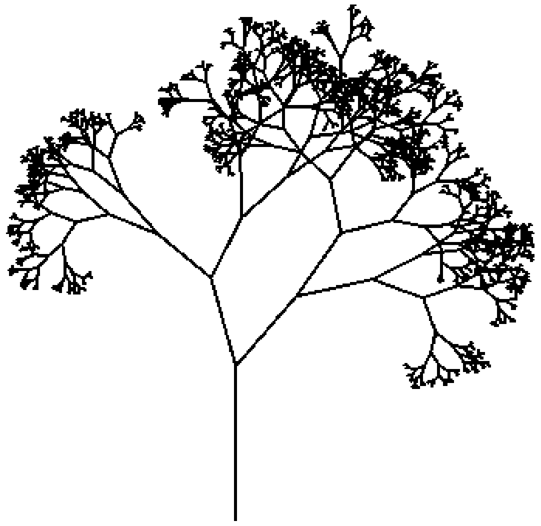
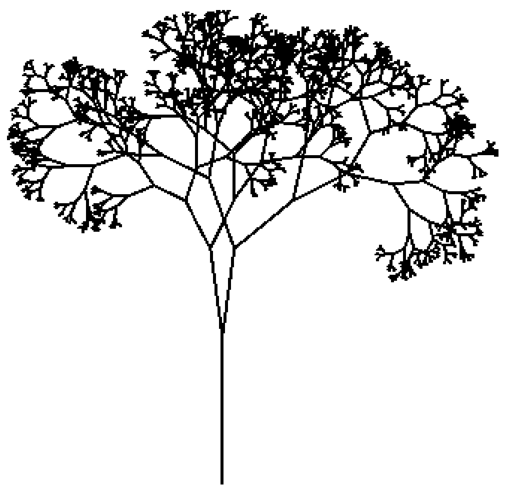
Another factor that we can adjust is the length of each branch. Currently, they shrink by a constant factor of **shrinkFactor** which is 0.6 in the examples above. However, we can shrink them by a random amount (within reason, of course). So, we can add or subtract a random amount from the **shrinkFactor**. It is easiest to simply multiply the **shrinkFactor** by some value, perhaps in the range from 1.0 to 1.5. That will allow some branches to be larger than 60% of their "parent" branch.

Here is the code as we now have it using a variable **bendAngle** and a new variable called **stretch** to allow randomly longer branches:

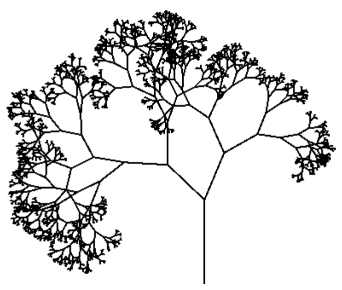
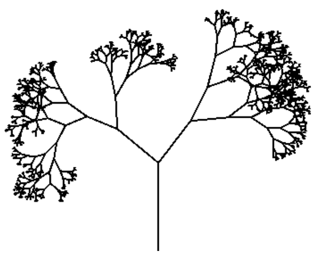
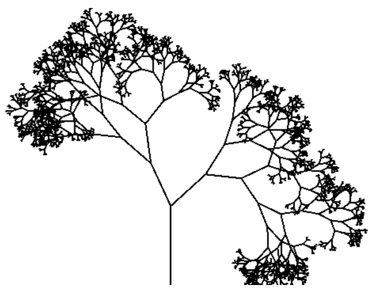
```
Function: drawNaturalTree
    length:     length of tree branch
    x, y        location to start drawing at
    dir:        direction to draw towards

    ... continued on next page ...
```

```
if (length < 2)
    return

bendAngle ← 5.6° + (random value from 0° to 45°)
endX ← x + length * cos(dir + bendAngle)
endY ← y + length * sin(dir + bendAngle+180°)
draw line from (x, y) to (endX, endY)
stretch ← 1 + (random value from 0 to 0.5)
drawNaturalTree (length * shrinkFactor * stretch, endX, endY, dir + bendAngle)

bendAngle ← 5.6° + (random value from 0° to 45°)
endX ← x + length * cos(dir - bendAngle)
endY ← y + length * sin(dir - bendAngle+180°)
draw line from (x, y) to (endX, endY)
stretch ← 1 + (random value from 0 to 0.5)
drawNaturalTree(length * shrinkFactor * stretch, endX, endY, dir - bendAngle)
```

Here are two examples of results due to this change:

Notice how tree is more "bushy". That is because the recursion goes further before it stops (i.e., it draws more branches). This is because the base case of the recursion stops only when the branch reaches a specific small size. Due to the randomness of the parameters, we may end up with some lopsided trees:

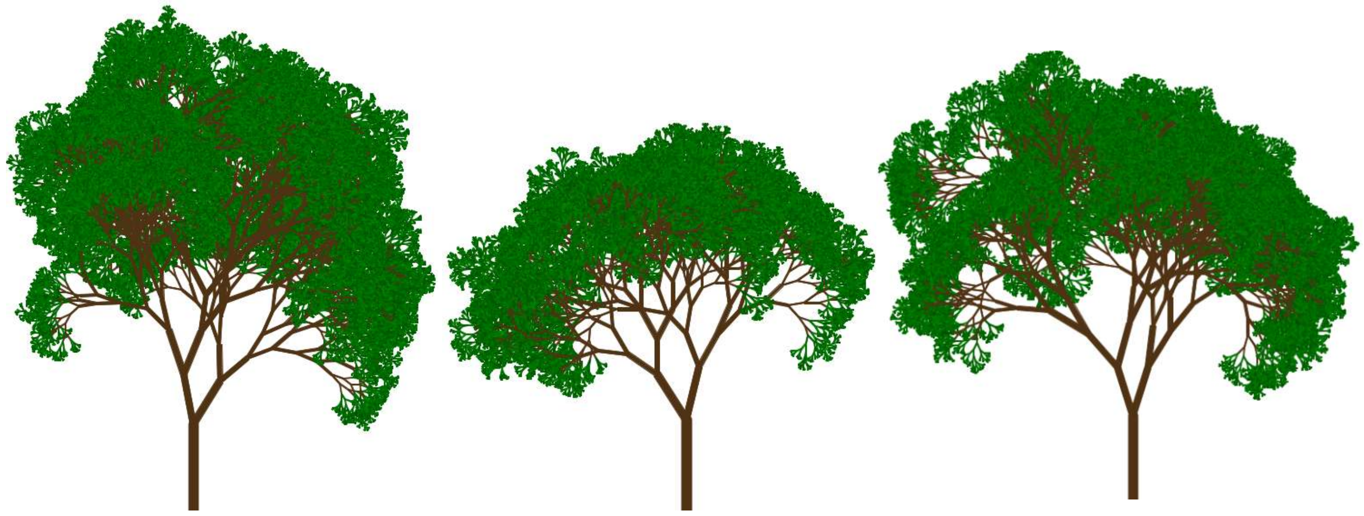
Of course, we can make everything even *bushier* if we increase the **shrinkFactor** to 65%:

All that remains now is to make the branches thicker and color them brown. For very small branches, we color them green:

```
Function: drawColorTree
        length:         length of tree branch
        x:              x location to start drawing at
        y:              y location to start drawing at
        dir:            direction to draw towards

1.      if (length < 2)
2.          return
2.      for b from -1 to 1 by 2 do {
3.          set line drawing width to length / 7
4.          if (length > 10) then
5.              set line drawing color to brown
6.          otherwise {
7.              set line drawing color to a random amount of green (e.g., 80 to 130)
8.              set line drawing width to 2 + (length / 7)
            }
9.          bendAngle ← 5.6° + (random value from 0° to 45°)
10.         endX ← x + length * cos(dir + b*bendAngle)
11.         endY ← y + length * sin(dir + b*bendAngle+180°)
12.         draw line from (x, y) to (endX, endY)
13.         stretch ← 1 + (random value from 0 to 0.5)
14.         drawColorTree (length * shrinkFactor * stretch, endX, endY, dir + b*bendAngle)
        }
```

Notice the **FOR** loop now which repeats exactly 2 times. This reduces our code since the code for drawing the left branch is the same as that for drawing the right branch, except that the angle offset is negative (represented by the value of **b**).

Here is the result (with **shrinkFactor** at 67% and **bendAngle** of (π/32 + random 0- π/6)):

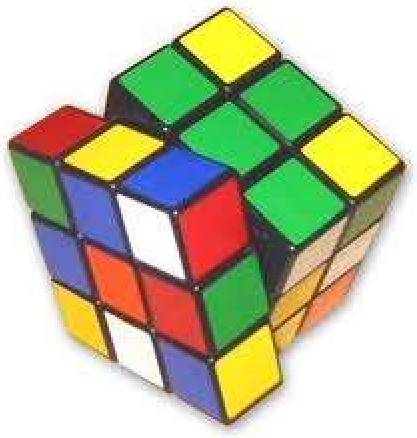


7.4 Search Examples

Sometimes, we use recursion to search for an answer to a problem. Searching for an answer recursively can often lead to a simpler, more elegant solution than writing search code that uses many loops and IF statements.

Example:

Consider writing code to solve a rubik's cube. The task may seem daunting at first ... but there are many systematic ways of solving the cube in steps or phases. Any non-recursive code that you would write could be quite complex, tedious and/or cumbersome. However, a recursive solution is quite simple.



Assume that the cube can be solved by making at most 100 quarter turns of one of the cube's sides (a reasonable assumption). Here is the idea behind the recursive solution:

> Try turning 1 of the 6 sides 1/4 of a turn, then try to solve the cube in 99 more turns, recursively. If the recursion comes back with a solution from the 99 additional moves, then the 1/4 turn that we made was a good one, hence part of the solution. If however, the recursion came back after 99 moves with an unsolved cube, then the 1/4 turn that we made was a bad choice ... undo it and then try turning one of the other 5 sides.

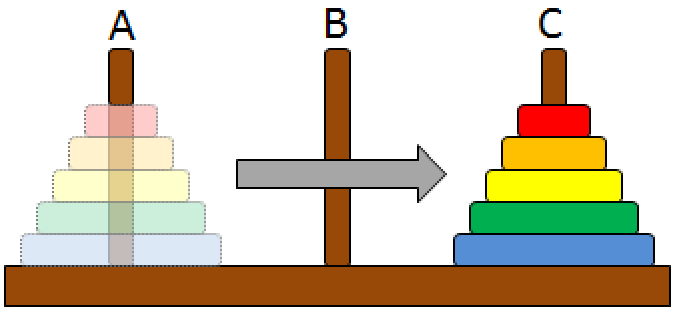
Here is the pseudocode for a function to solve the cube recursively in **n** steps, returning **true** if it has been solved, or **false** otherwise:

```
Function: SolveRubiksCube
        cube:   the cube data structure
        n:      maximum number of steps remaining to solve the cube

1.      if (cube is solved) then
2.              return true
3.      if (n is 0) then
4.              return false
5.      for each side of the cube s from 1 to 6 do {
6.              turn side s 1/4 turn clockwise
7.              recursiveAnswer ← SolveRubiksCube(cube, n-1)
8.              if (recursiveAnswer is true) then
9.                      return true
10.             turn side s 1/4 turn counter-clockwise          // i.e., undo the turn
11.     }
12.     return false
```

That is the solution! It is quite simple. Do you see the power of recursion now ? Of course, this particular solution can be quite slow, as it blindly tries all possible turns without any particular strategy. Nevertheless, the solution is simple to understand. It would be difficult to produce such a simple function to solve this problem without using recursion.

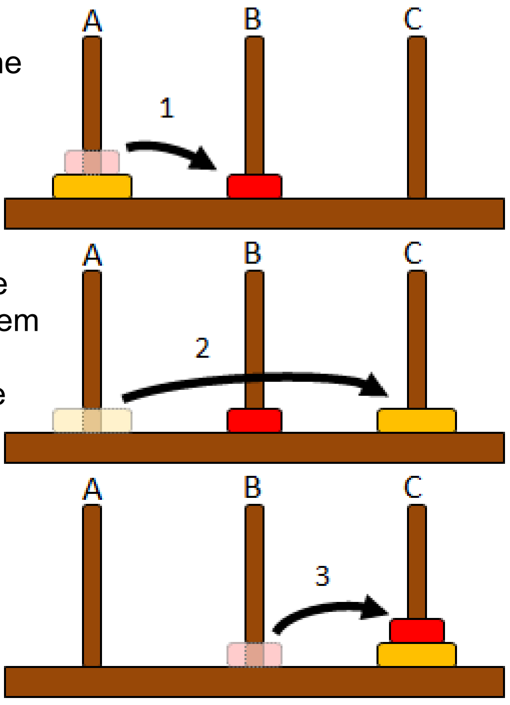
Example:

Another problem that is naturally solved recursively is that the **Towers of Hanoi** puzzle. In this puzzle game, there are three pegs and a set of **n** disks of various sizes that can be moved from one peg to another. To begin, all **n** pegs are stacked on the first peg in order of increasing sizes such that the largest disk is at the bottom and the smallest is at the top. The objective of the game is to move the entire tower of disks to a different peg by moving one disk at a time and doing so with a minimum number of steps.



There are just two rules:

1. disks must be moved from one peg to another, not placed aside on the table
2. a disk can never be placed onto a smaller-sized disk.

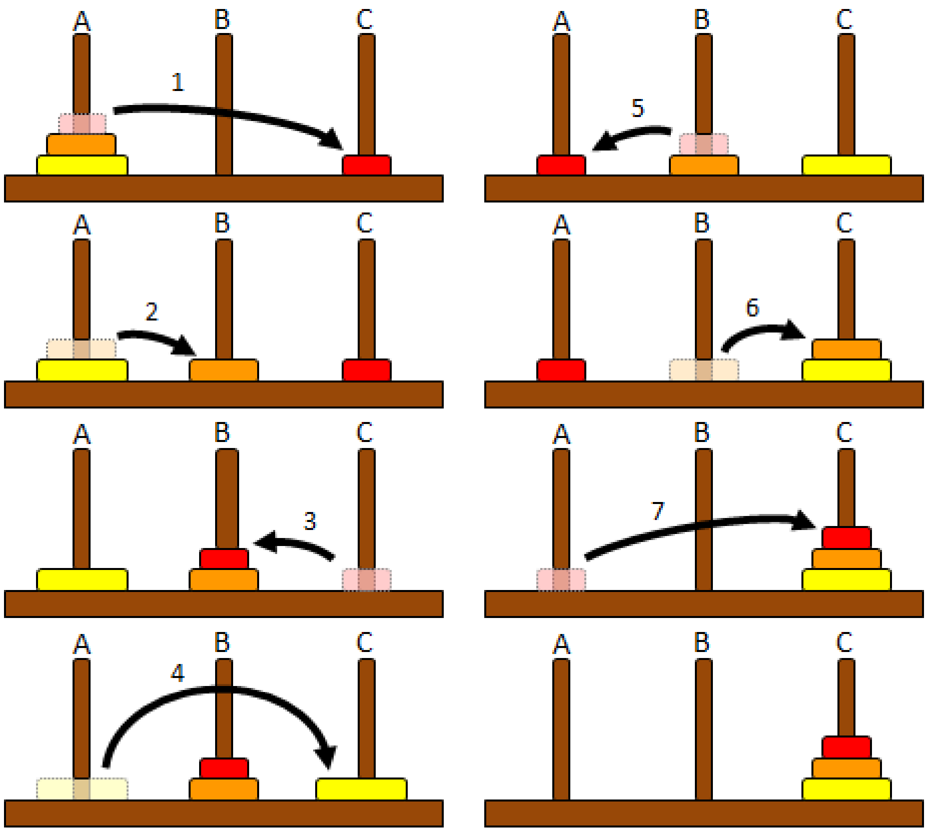


To approach this problem recursively, we must understand how to break the problem down into a smaller problem of the same type. Since all we are doing is moving disks, then a "smaller" sub-problem would logically just be moving less disks. For example, if **1** disk is moved, then only **n-1** disks remain to be moved ... which a smaller problem.

So what then would be the base case ? Well ... what is the simplest number of disks to move ? Zero. A simple problem is also the case where **n = 1**. In that case, we simply pick up the disk and move it. For values of n greater than **1**, we now have to worry about moving some disks out of the way. Here is the solution when **n = 2** →

Notice that after the small red disk is moved to peg B, then we have the smaller/simpler problem of moving the one larger disk from peg A to peg C. Finally, we have the simpler/smaller problem of moving the red disk from peg B to peg C and we are done.

It may be hard to see the recursion, at this point, so let us look at the solution for **n = 3**:



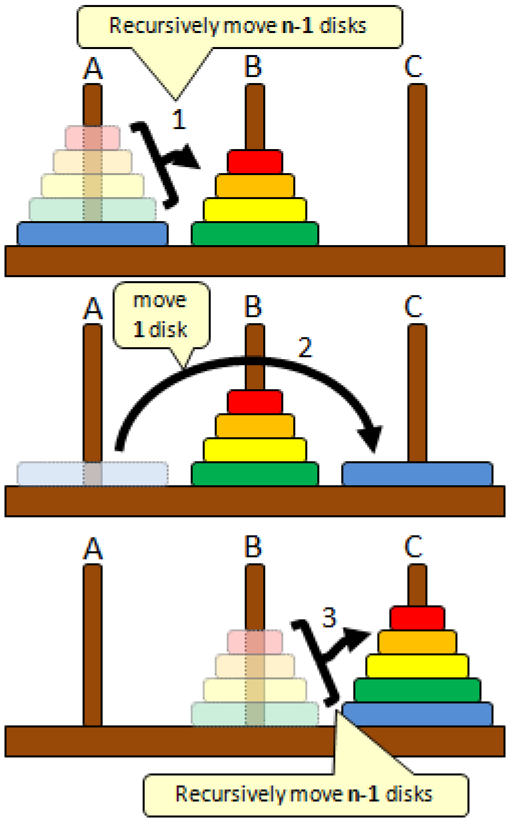
You may notice that steps 1 to 3 are basically the same as the problem where **n = 2**, except that we are moving the disks from peg **A** to peg **B** instead of to peg **C**. Then step 4 simply moves the 1 disk from peg **A** to peg **C**. Finally, we are left with the recursive sub-problem again of moving 2 disks from peg **B** to peg **C**.

So, the recursive solution of moving **n** disks is based on this idea:

1. Move the **n-1** smaller disks out of the way to the spare peg **B**

2. Move the large disk to peg **C**

3. Move the **n-1** smaller disks to peg **C**



As you can see above, there are two recursive sub-problems corresponding to moving **n-1** disks out of the way first, and then solving the **n-1** problem once the largest bottom disk is in place.

So, we can now write the code by defining a procedure that indicates the **peg to move from**, the **peg to move to**, the **extra peg** and the **number of disks to move**.

```
Procedure: TowersOfHanoi
        pegA:     the peg to move from
        pegC:     the peg to move to
        pegB:     the extra peg
        n:        number of disks to move

1.      if (n is 1) then
2.              Move 1 disk from pegA to pegC
        otherwise {
3.              TowersOfHanoi(pegA, pegB, pegC, n-1)
4.              Move 1 disk from pegA to pegC
5.              TowersOfHanoi(pegB, pegC, pegA, n-1)
        }
```

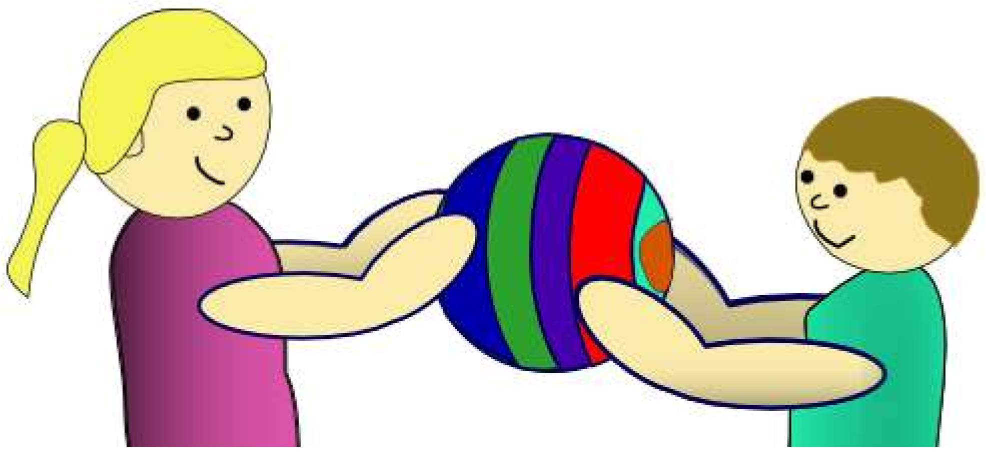
As you can see, the problem is expressed in a simple manner with only about 5 lines. Notice that there was no need to check if **n = 0** since that would represent a problem that is already solved.

Chapter 8

Shared Data

What is in This Chapter ?

This chapter discusses the notion of **sharing data** in your programs. It explains situations in which sharing of data can be useful to simplify your program and be memory efficient. The chapter also explains how sharing of data is sometimes necessary in order for your program to run and then gives an example showing potentially unpleasant consequences of not being careful when dealing with shared data. Lastly, a large **graph editor** example is given that incorporates shared data and brings together many of the concepts that you have learned throughout this course.



8.1 Sharing Data Can Be Useful

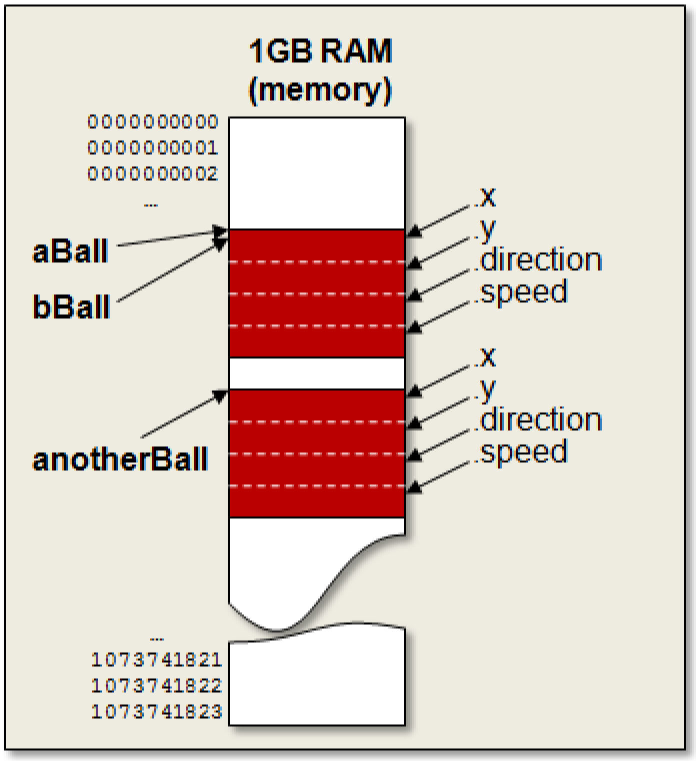
Recall in our discussion of variables that a variable is ***bound to*** *(i.e., attached to)* a value when we assign something to it using the **=** operator.

```
x = 100;
name = "Bob";
```

When we create our own data structures (i.e., objects) we need to remember that the data that makes up the object (i.e., the object's attributes) is stored in the computer's memory. The object itself is simply just a reference (i.e., a pointer) to the location in memory where the object's attributes are actually being stored. Each object created is a unique reference (i.e., memory location in the computer's memory).

Consider creating two ball objects as follows:

```
Ball   aBall;
Ball   anotherBall;

aBall = new Ball();
anotherBall = new Ball();
```



Since each ball is stored in a different memory location, each has its own set of unique attribute values. That is, each ball has its own (**x**,**y**) location, **direction** and **speed**.

Sometimes, however, we can end up in a situation in which two balls share the same memory location. Consider adding another ball variable as follows:

```
Ball   bBall;

bBall = aBall;
```

Now, **bBall** and **aBall** are actually references to the same object ... that is ... they are pointing to the same memory location and therefore share the same attributes.

Having two objects share the same memory can be advantageous since they can share the same information together, using up less memory space. For example, consider a "family car" in which many members of the same family drive the exact same vehicle, thereby reducing the need to buy another car and saving space in the laneway.

Of course, sharing does have some disadvantages. For example, if someone borrows the car and brings it back with no gasoline, then it affects the next person who will use the car. Also, the seat and mirrors may all need to be adjusted for the next driver. Worse yet, if one person in the family crashes the car, then the car is ruined for everyone.

When programming, it is important to understand when data structures are being shared so that efficient programs can be written, while ensuring that the data from one object does not interfere with the data from other objects unexpectedly.

Example:

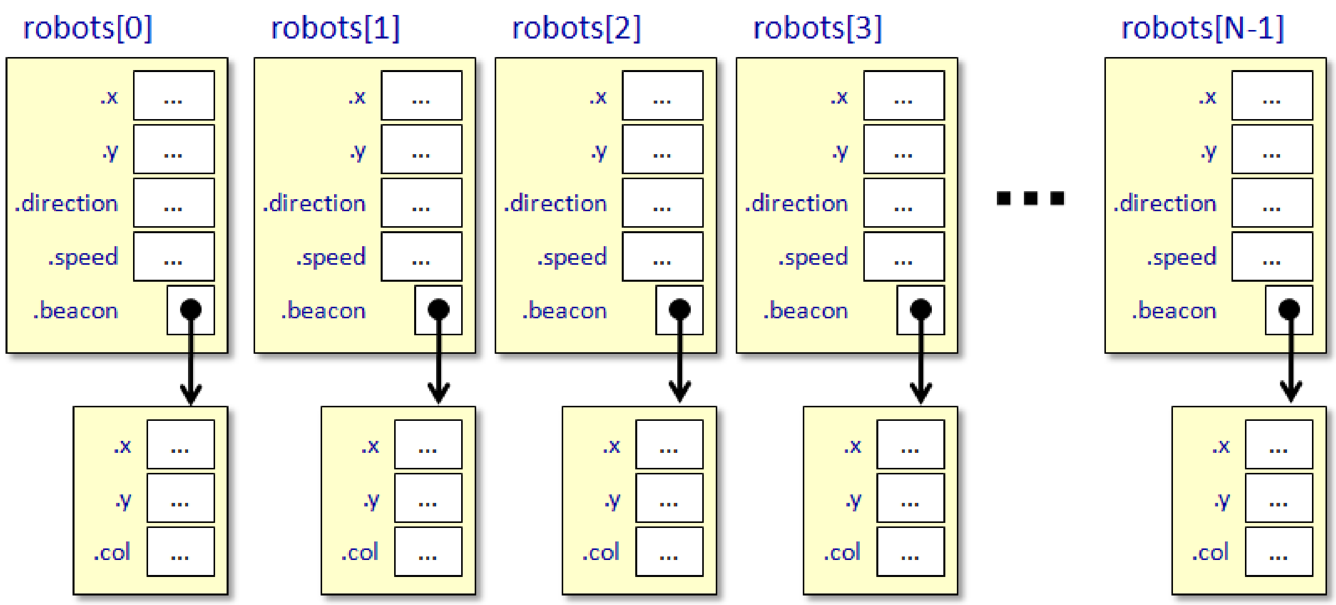
Consider simulating a swarm of insect-like robots that are attracted towards a beacon such as a light source. We can define a beacon as having a particular (x,y) location on the window and perhaps a randomly-chosen color:

```
class Beacon {
  float        x, y;
  color        col;

  Beacon(float bx, float by) {
    x = bx;
    y = by;
    col = color(55+random(200),55+random(200),55+random(200));
  }
}
```

Now to simulate the robots, let us define robot's as having an (**x**, **y**) location, a **direction** and a **speed**. We will also assign a **Beacon** to each robot as some particular location to head towards. The definition is very similar to the **Ball** data structure that we defined previously:

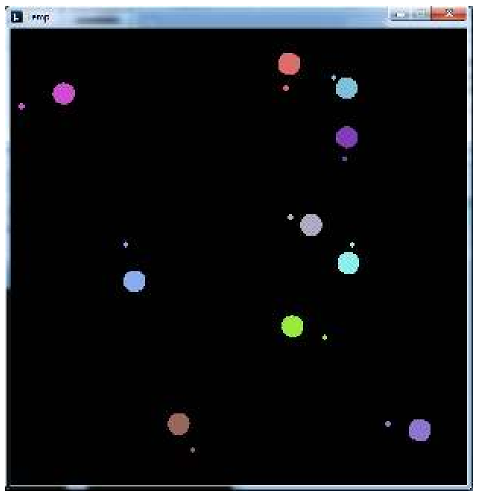
```
class Robot {
  float     x, y;               // location of the robot at any time
  float     direction;          // direction of the robot at any time
  float     speed;              // the robot’s speed
  Beacon    beacon;             // the beacon to head towards

  Robot(float rx, float ry) {
    x = rx;
    y = ry;
    direction = random(TWO_PI);     // a random direction
    speed = 3 + random(4);          // a random speed from 3 to 6
    beacon = null;                  // not set yet
  }
}
```

Notice that each robot now stores their own **Beacon** object (i.e., their own place to head towards). We can draw the data structures as follows ... notice how each robot has its own **Beacon** object, allowing them all to head in different directions:



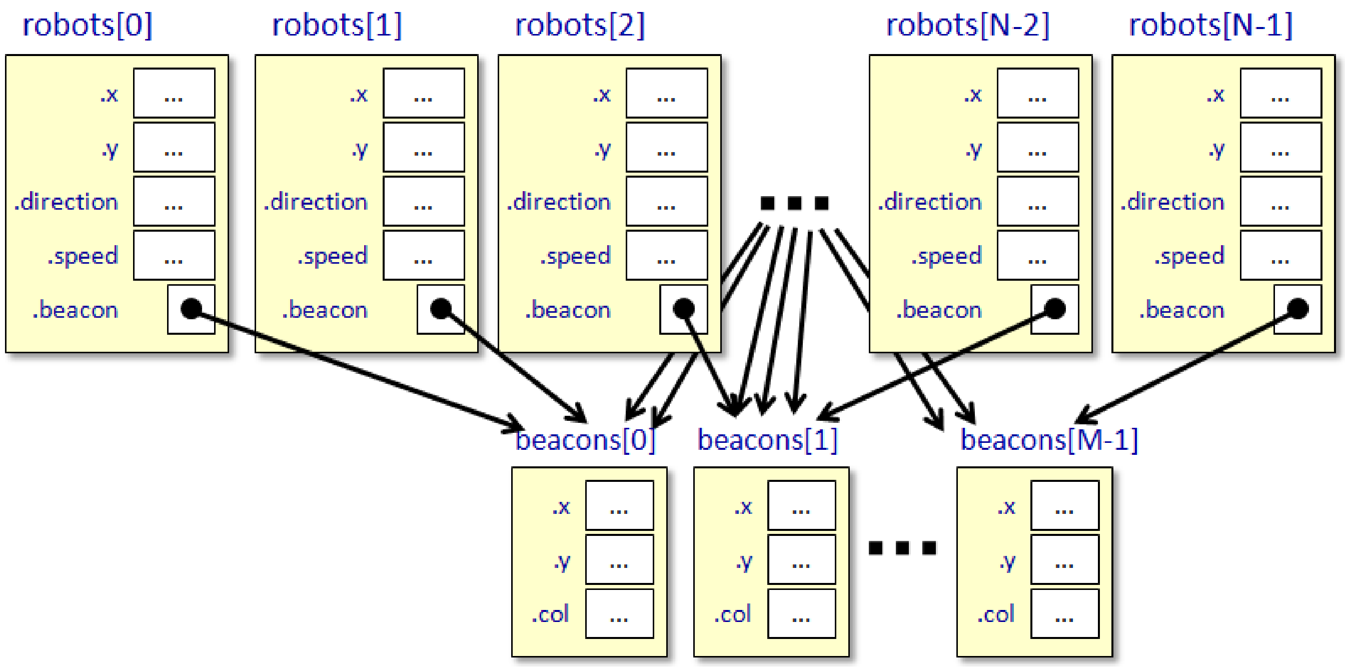
Here is the additional code required to create and display the robots and their beacons:

```
Robot[]      robots;                // a bunch of robots
final int    ROBOT_RADIUS = 2;      // a robot's radius (in pixels)
final int    BEACON_RADIUS = 15;    // a beacon's radius (in pixels)

void setup() {
  size(600,600);

  // Make some robots with unique beacons
  robots = new Robot[10];
  for (int i=0; i<robots.length; i++) {
    robots[i] = new Robot(width/2, height/2);
    robots[i].beacon = new Beacon(random(width), random(height));
  }
}

void draw() {
  background(0,0,0);
  for (int i=0; i<robots.length; i++) {   // draw all robots and their beacons
    draw(robots[i]);
    draw(robots[i].beacon);
    move(robots[i]);  // explained later
  }
}

void draw(Robot r) {
  fill(r.beacon.col);  // use beacon's color
  stroke(0,0,0);
  ellipse(r.x, r.y, 2*ROBOT_RADIUS,2*ROBOT_RADIUS);
}

void draw(Beacon b) {
  fill(b.col);
  stroke(0,0,0);
  ellipse(b.x, b.y, 2*BEACON_RADIUS,2*BEACON_RADIUS);
}
```



Notice how each robot is colored to match its beacon location's color. The **move()** procedure will be explained later. For now, assume though that it steers the robot towards its beacon location.

In the code above, each robot has its own unique beacon to head towards. Now we will adjust the code to allow multiple robots to head towards the same beacon, thereby causing a swarm-like simulation. We will alter the code so that instead of having a unique beacon for each robot, we will generate only a few beacons and have the robots share the same exact **Beacon** object as their target destination. Here is how the data structure will change:



There will be much less **Beacon** objects created since multiple robots will share the same one. Is there any advantage to having less/shared **Beacons** ?

One important advantage to having shared beacons like this is that memory storage requirements are reduced. Imagine for example, that it takes **4** bytes each to store the **x** and **y** values of the beacon location and an additional **3** bytes to store the **color**. That means, each beacon requires a minimum of **11** bytes of storage. If we simulated **N** robots each storing their own unique beacons, this would require ((**11**+**4**) x **N**) bytes of storage in memory just to store the beacons (the extra **4** bytes storing the pointer to the beacon in the memory).

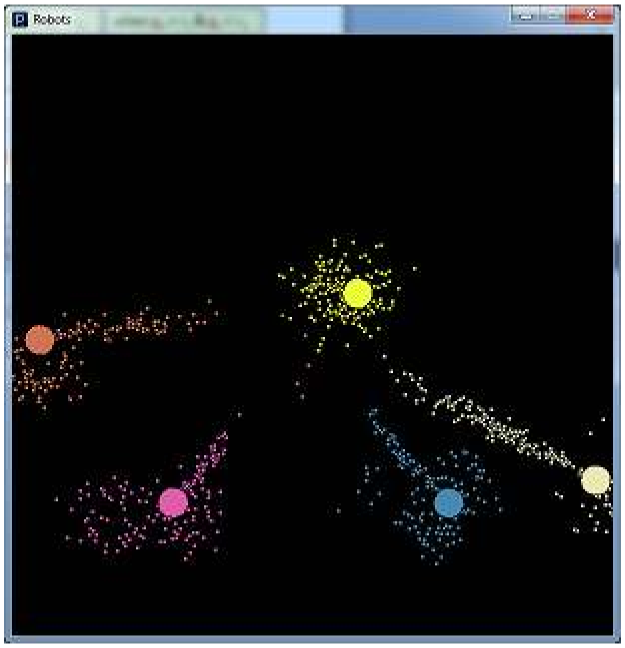
Now, assume that we create **M** beacons. This would require (**11** x **M**) bytes of storage. If each robot now kept only a single pointer (i.e., **4** bytes) to one of the **M** beacons, the total storage requirements for the beacon storage would be ((**4** x **N**) + (**11** x **M**)) bytes.

This seems a little abstract. Assume then that we have **5** beacons and **1000** robots. Then this works out to be **15,000** bytes for the non-shared beacon version and **4,055** bytes for the shared beacon version. That is a significant difference of less than **1/3** of the storage space requirements!!

This is often the reason for having shared data and shared objects ... to reduce storage space requirements for the program. However, in addition to reduced storage space, there is another important advantage to having shared beacons. If the beacon's location was to change (e.g., manually moved by the user), we would simply need to alter the (**x,y**) location of the single **Beacon** object, and since all same-swarm robots share this same **Beacon** object in memory, they will all be able to access (i.e., when heading towards) the newly changed (**x,y**) location immediately. Therefore, by changing a single variable (e.g., a beacon's **x** location), we are automatically modifying the behavior of multiple robots. This allows us to simulate the objects efficiently. Conversely, if we had stored a unique beacon object within each robot (i.e., non-shared beacons), we would need to find all robot's that share that beacon and update all of their **x** coordinates. This would be a slower process since it would require us to loop through all robots, checking their beacons for a match as to which ones the user is trying to move.

To verify this, consider altering the setup code to produce a newly-defined array of 5 beacons and then adjust the robot's to choose one of these 5 beacons as their destination. We also should adjust the **draw()** procedure to draw the beacons separately:

```
...
Beacon[]    beacons;                // a bunch of beacons

void setup() {
  ...
  beacons = new Beacon[5];
  for (int i=0; i<beacons.length; i++)
    beacons[i] = new Beacon(random(width), random(height));

  // Make some robots with shared beacons
  robots = new Robot[1000];   // lots of robots
  for (int i=0; i<robots.length; i++) {
    robots[i] = new Robot(width/2, height/2);
    robots[i].beacon = beacons[int(random(beacons.length))];  // choose random one
  }
}

void draw() {
  background(0,0,0);

  // draw all robots
  for (int i=0; i<robots.length; i++) {
    draw(robots[i]);
    move(robots[i]);  // explained later
  }

  // draw all beacons separately
  for (int i=0; i<beacons.length; i++)
    draw(beacons[i]);
}
```



Notice how each robot is now assigned a random beacon from the **beacons** array. The program will now allow 5 groups of robots to head towards one of the 5 beacons in a swarm-like fashion.

For added enjoyment, we can add the following to allow beacons to be grabbed and moved around on the screen:

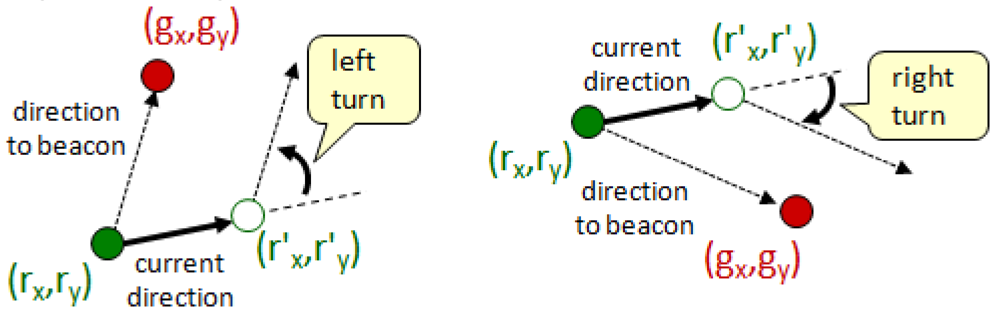
```
Beacon       grabbed;     // the beacon that has been grabbed

void mousePressed() {
  for (int i=0; i<beacons.length; i++) {
    if (dist(beacons[i].x, beacons[i].y, mouseX, mouseY) < BEACON_RADIUS) {
      grabbed = beacons[i];
      return;
    }
  }
}
void mouseDragged() {
  if (grabbed != null) {
      grabbed.x = mouseX;
      grabbed.y = mouseY;
  }
}
void mouseReleased() {
  grabbed = null;
}
```

As the beacon is moved around, you will see a subset of the robots following it around. Clearly, by allowing robot's to share **Beacon** objects we gain the advantages of saving storage space and also easy of updates when we change a beacon's location.

The only missing component of our code is the **move()** procedure...which is responsible for moving the robot towards the beacon. Recall from our ball-moving example that to move a ball (or robot) forward, we simply apply trigonometry using the robot's location and direction:

```
void move(Robot r) {
      r.x = r.x + int(r.speed*cos(r.direction));
      r.y = r.y + int(r.speed*sin(r.direction));
}
```

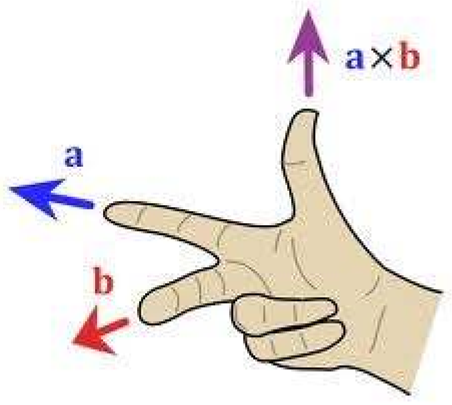
The code above simply moves the robot forward in its current direction. The harder part is to determine and update the direction of the robot so that it heads towards the beacon location. All we need to do is to look at the where the robot is heading and decide whether or not it needs to turn right or left to get towards the beacon:



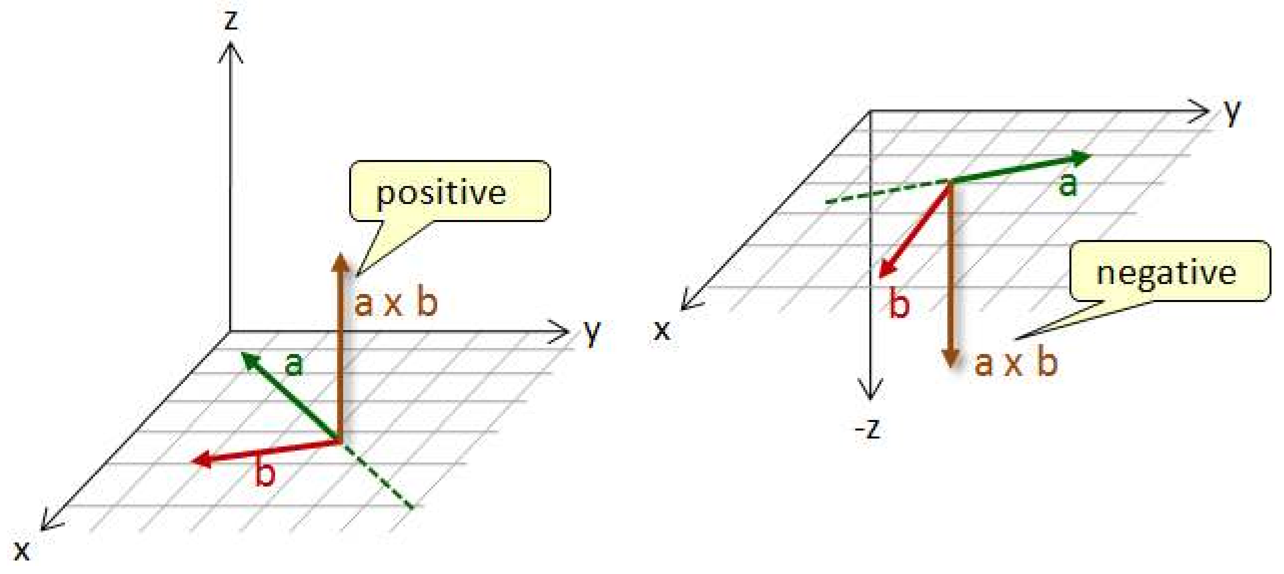
Above, ($\mathbf{r_x}$, $\mathbf{r_y}$) is the robot's current location and ($\mathbf{r'_x}$, $\mathbf{r'_y}$) would be the robot's next location if it were to travel in its current direction without turning (for this next location we could simply use the new value for **r.x** and **r.y** that we computed in our code above).

The beacon location is represented as (g_x,g_y). In order to determine whether or not the robot should take a left or right turn to get to the beacon from its current direction, we can examine the type of turn from (r_x, r_y) → (r'_x, r'_y) → (g_x,g_y). If this is a left turn, the robot should turn left. If it is a right turn the robot should turn right. If it is a straight line, the robot will need to either move straight ahead, or straight backwards (depending on where the beacon location is).

To determine the type of turn, we can make use of the ***cross product*** of the vector from (r_x, r_y) → (r'_x, r'_y) and the vector from (r_x, r_y) → (g_x,g_y). You may recall that the cross product **a X b** is a vector that is perpendicular to the plane containing the two vectors **a** and **b**. The cross product will either be positive, negative or zero. You can visualize the cross product by using the right-hand rule as shown in the picture here.



If the cross product is positive, this indicates a left turn. If negative, then a right turn. If the cross product is zero, then there is no turn (i.e., the two vectors form an angle of 180°).



The cross product can be computed as follows:

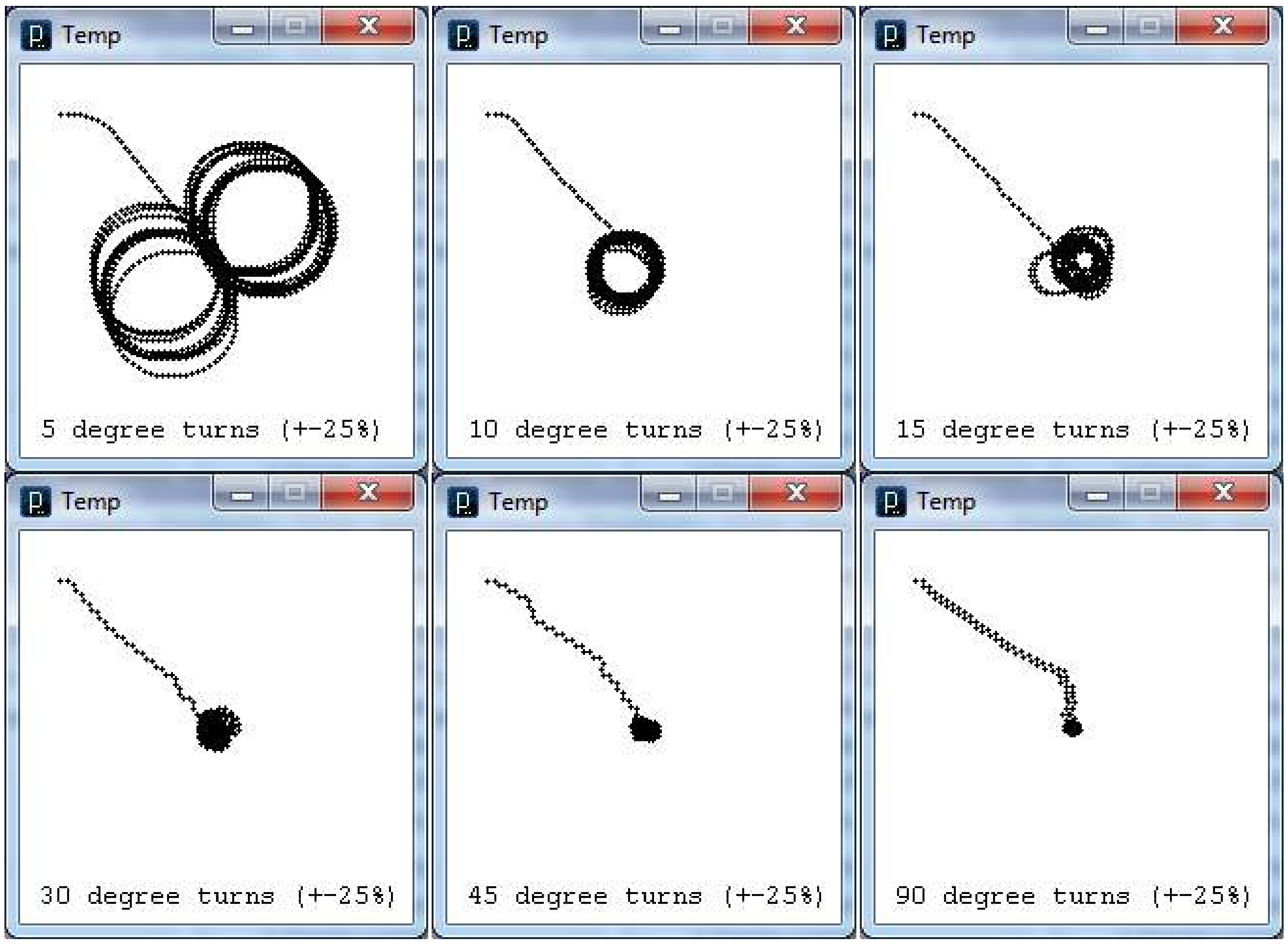
$$\text{crossProduct} = (r'_x - r_x)(g_y - r_y) - (r'_y - r_y)(g_x - r_x)$$

So, we can just plug this into our program and check for the sign of the result. Once we determine whether to turn left or right, we simply add some degree amount to the robot's direction. If we use a large degree amount (e.g., 90°) then the robot will make only right-angled turns ... not too realistic. If we use a smaller amount (e.g., 45°), then the robot will make sharp turns and will orient towards the beacon quickly. Smaller amounts (e.g., 30° or 15°) will provide a more smooth motion. Very small amounts (e.g., 1°) will cause the robot to always makes large arcing motions, taking a while to orient towards the beacon.

To make the swarm more realistic, instead of having a fixed turn amount, we could adjust it to have a ±12.5% error as follows, given a desired turn amount of **θ**:

$$\text{amountToTurn} = \theta + \text{random}(\theta/4) - (\theta/4)/2$$

As a result, here is the kind of movement pattern that a robot would produce for various values of **θ** (note that the beacon location is not displayed, but is in the center if the window):



```
void move(Robot r) {
  float nextX, nextY, crossProduct;

  nextX = r.x + int(r.speed*cos(r.direction));
  nextY = r.y + int(r.speed*sin(r.direction));

  crossProduct = (nextX - r.x)*(r.beacon.y - r.y)-(nextY - r.y)*(r.beacon.x - r.x);
  r.x = nextX;
  r.y = nextY;
  if (crossProduct < 0)  // try changing to:  if((crossProduct<0)&&(random(1)<0.6))
    r.direction -= (ANGLE + random(ANGLE/4) - (ANGLE/8))/180*PI;
  else
    r.direction += (ANGLE + random(ANGLE/4) - (ANGLE/8))/180*PI;
}
```

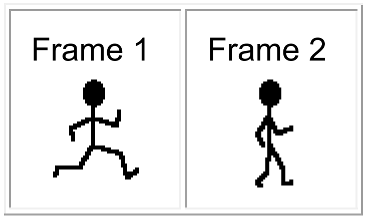
8.2 When Shared Data is Necessary

In some cases, it is absolutely necessary to share data in order to have a working program. Consider the area of computer animation. To animate something simple (i.e., 2D) in a program, programmers often use what are called ***sprites***.

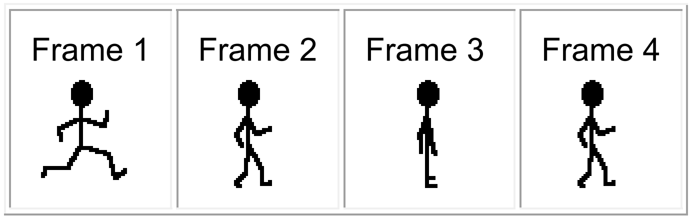
*A **sprite** is a 2D image that it integrated into a scene to form an animated character.*

Sprites are often displayed and animated by drawing a set of pictures in sequence. Each of the individual pictures, called ***frames*** represent the different "movements" of the object to be animated. So, by displaying these frames in sequence, we achieve animation.

For example, consider a stick person walking. We can do this with only two frames and just swap between them:

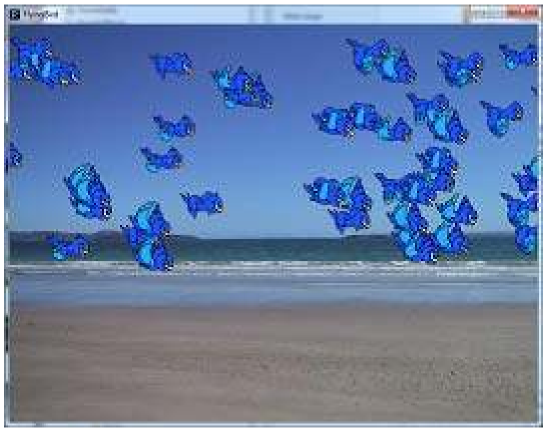


The animation, however, would have the undesirable characteristic of being very "jumpy", not very smooth. The problem is that there is no smooth transition between the frames. We can make a big improvement just by introducing one more picture and duplicating the 2nd frame twice to produce a 4 frame sequence:



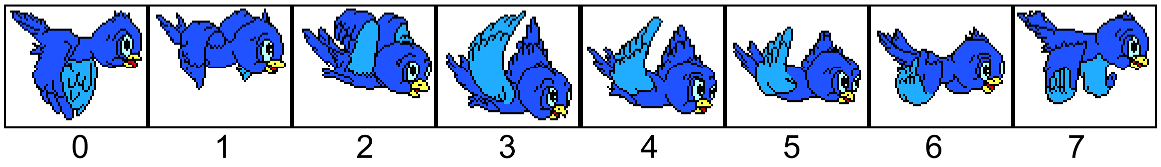
This animation would appear much smoother, but if we display the frames at the same rate, the person would appear to walk much slower. Assuming that we display 1 frame every 1/4 second, the 2-frame case would take 1/2 second to complete a cycle while in the 4-frame case, would take a full second. For the 4-frame case we can just reduce the inter-frame display delay to 1/8 of a second and the speed will then appear to match the 2-frame case. The number of frames that can be displayed in a second is known as the ***frame rate***. So, by using a delay of 1/8 seconds between frames, we have a frame rate for the animation of 8 frames per second (fps).

Example:



Consider a program that simulates birds flying across a beach scene. To make this look somewhat realistic (from a cartoon perspective), we would need to have unique pictures (i.e., frames) to display that represent the various "poses" of the bird as it flies.

We will make use of exactly 8 different poses (i.e., frames) for the birds as shown below. Each pose must be stored as a unique **.gif** image in a subfolder of our processing program's folder called **data**. The indices below the images represent the frame number:



We will need to maintain information for each bird in regards to its location on the screen, its speed and perhaps which frame is being shown at any time. Here is how we can define such a **Bird** data structure:

Define a bird object as follows:

```
class Bird {
  float      x, y;              // bird's coordinate on the screen
  float      speed;             // bird's speed in pixel movements per frame
  PImage[]   images;            // frames/pictures of the bird in various poses
  int        currentFrame;      // current frame of the bird

  Bird(float bx, float by) {
    x = bx;
    y = by;
    speed = random(10) + 5;          // random speed from 5 to 15 pixels/frame
    currentFrame = int(random(8));   // start with a random picture
    images = new PImage[8];          // array to hold 8 pictures
    for (int i=1; i<=8; i++)         // load up all 8 pictures for this bird
      images[i-1] = loadImage("Birdx" + i + ".gif");
  }
}
```

We can then create an array of **Bird** objects and display them on the screen as shown in the following code. Note that the beach image is a separate image that is displayed as a background picture. The code will display 50 birds at random locations on the window at some random pose. Even though the birds are displayed, they are not moving...they seem still. The **frameRate** is the number of times per second that the **draw()** procedure is called. It is set to 20 and this will be fast enough to animate the birds, as we will see later. If it is not set, then the **frameRate** will be too fast because the default rate is 60 fps.

```
Bird[]        birds;
PImage        beach;

void setup() {
  size(800,600);
  birds = new Bird[50];
  for (int i=0; i<8; i++)
    birds[i] = new Bird(random(width),random(height/2));
  beach = loadImage("beach.jpg");
  frameRate(20);
}

void draw() {     // This is called repeatedly at a fixed frameRate
  image(beach, 0, 0);
  for (int i=0; i<8; i++)
    draw(birds[i]);
}

void draw(Bird b) {
  image(b.images[b.currentFrame], b.x, b.y);
}
```

To animate the birds, we must now do two things: (1) Have them vary their images, and (2) have them move horizontally. We can update the **draw()** procedure by adding a **move()** procedure as follows:

```
void draw() {     // This is called repeatedly at a fixed frameRate
  image(beach, 0, 0);
  for (int i=0; i<8; i++) {
    move(birds[i]);
    draw(birds[i]);
  }
}
```

The **move()** procedure must move the bird forward and this can be done simply by adding the bird's speed to the x coordinate, making sure that when it reaches the right side of the window, we cause a wraparound to the left side again. To do this, we simply set the **x** value to an amount which is **-width** of the bird image so that the bird slowly flies back onto the window again from the left. To change the bird's image, we simply increment the **currentFrame**, making sure that when it reaches 8, we set it back to 0 again ...using the modulus operator:

```
void move(Bird b) {
  // Move the bird 10 pixels forward
  b.x = b.x + b.speed;

  if (b.x > width)
    b.x = -102; // 102 is the width of the Bird image

  b.currentFrame = (b.currentFrame + 1) % 8;
}
```

Now the birds will appear to be flying nicely. Try changing the array to make **500** birds. Depending on your computer's memory ... the code may or may not run. My computer produced an error indicating that I would need to increase the memory setting in the **Preferences** under the **File** menu.

The problem is simple. The current code loads 8 images for each bird. Therefore, 500 birds would require (500 x 8 = 4000) images to be loaded and stored in the program. However, it is easily seen that the birds are all using the same 8 image files. Therefore, instead of having each bird store their own images, it makes more sense to simply load the 8 images and have the birds "share" the images. We can add this to the program:

```
PImage[]      birdFrames;
```

and then add this to the **setup()** procedure:

```
birdFrames = new PImage[8];
for (int i=1; i<=8; i++)
      birdFrames[i-1] = loadImage("Birdx" + i + ".gif");
```

Then we can remove the **images** from the **Bird** object:

```
class Bird {
  float      x, y;
  float      speed;
  int        currentFrame;

  Bird(float bx, float by) {
    x = bx;
    y = by;
    speed = random(10) + 5;
    currentFrame = int(random(8));
  }
}
```

Finally, we make changes to the **draw(Bird b)** procedure:

```
void draw(Bird b) {
  image(birdFrames[b.currentFrame], b.x, b.y);
}
```

Now, you may even be able to add up to 50,000 birds without problems...although the program would become very slow ;)

So in this example, we see that sharing is sometimes necessary in order to reduce memory space requirements and have a running program. There are also examples in which sharing is NOT even desired at all...

8.3 Separating Shared Data Again

Example:

Even though sharing objects may be to our advantage, there are some situations where we do not want shared objects. For example, sometimes we need to make copies of objects and we would like the copy to be a truly unique copy so that we can leave the original intact and safe at all times. We may think of making photocopies of important documents so that the original document can be stored away safely.

Consider the following function which creates and returns a copy of a list:

Function: makeCopy(aList)

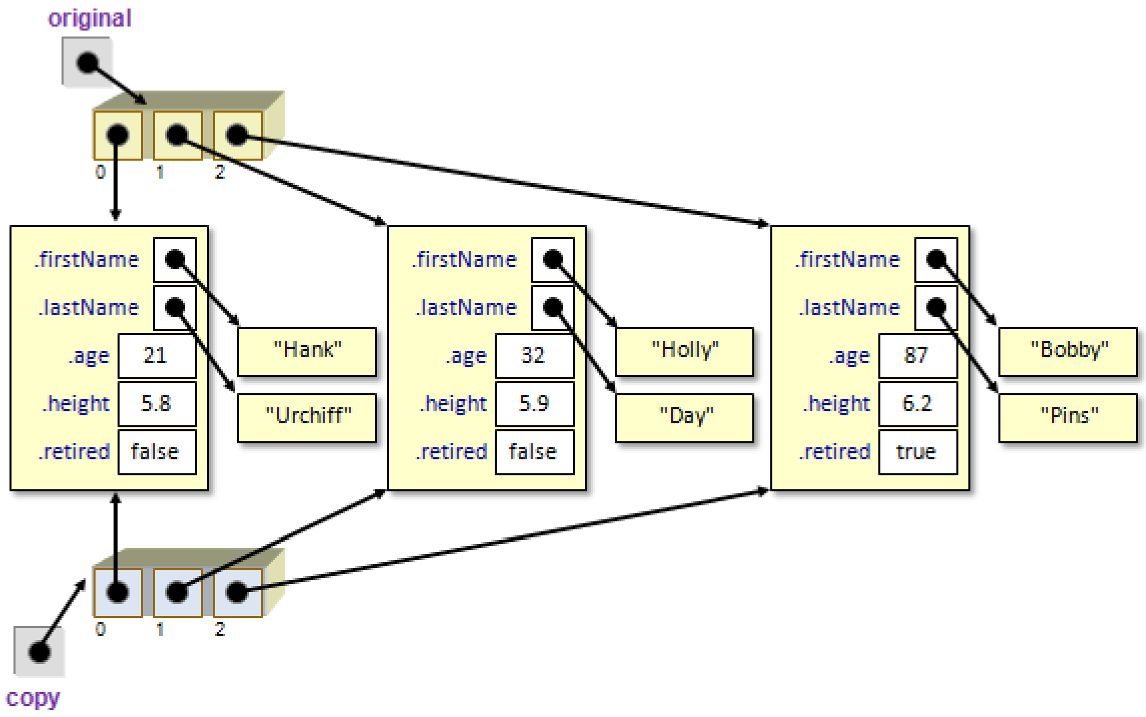
```
1.    anotherList ← new array with capacity aList.length
2.    for index i from 0 to aList.length - 1
3.        anotherList[i] ← aList[i]
4.    return anotherList
```

The code produces a new array containing the objects from the original list. Consider testing this function with an array of **Person** objects, which have a firstName, lastName, age, height and retiredStatus as follows:

Algorithm: CopyTest1

```
1.    original ← new array with capacity 3
2.    original[0] ← new Person("Hank", "Urchif", 21, 5.8, false)
3.    original[1] ← new Person("Holly", "Day", 32, 5.9, false)
4.    original[2] ← new Person("Bobby", "Pins", 87, 6.2, true)
5.    copy ← makeCopy(original)
```

Here is what we accomplish with this code …



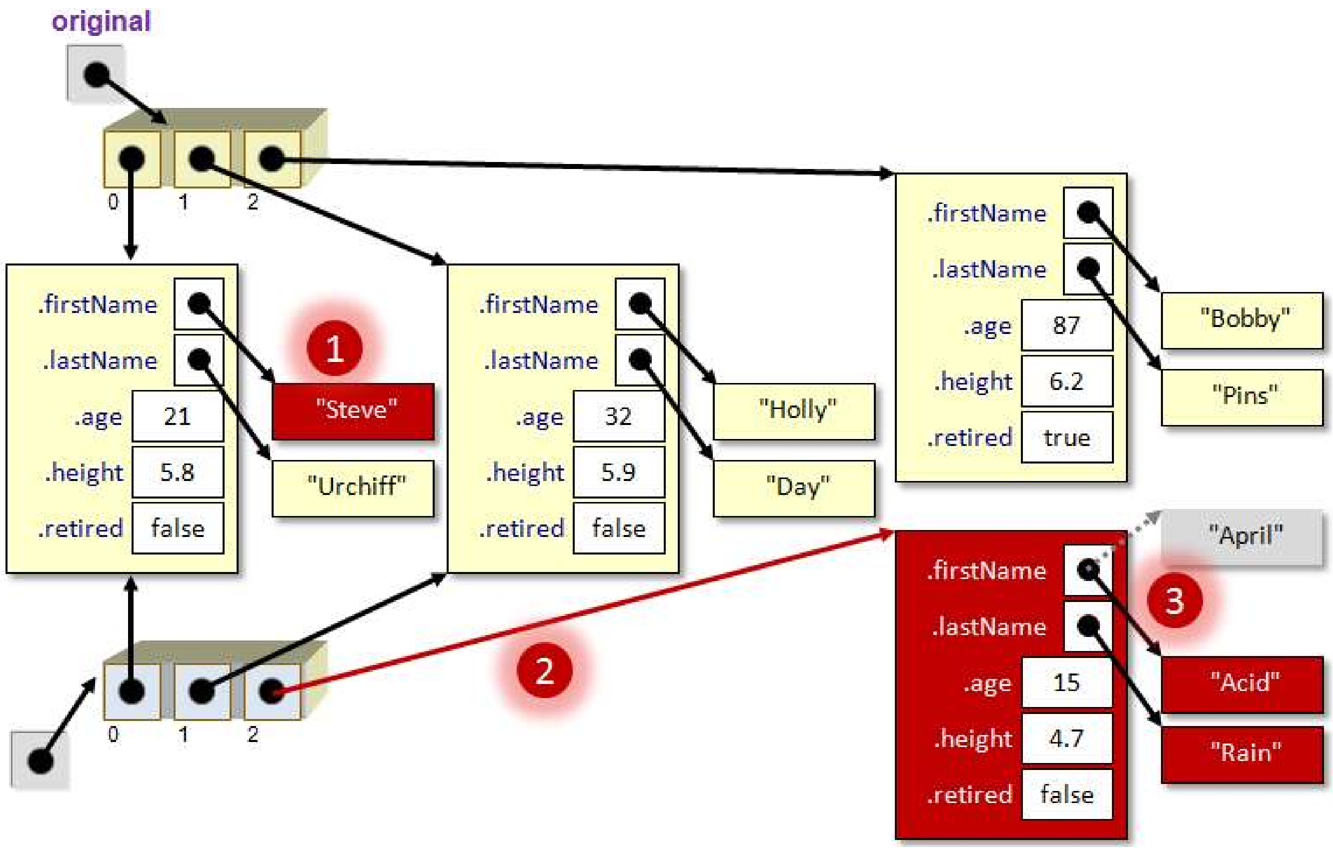
Notice that when we add the **Person** objects to the copy, they are actually shared between the two arrays. This is known as a ***shallow copy***. The advantage is that we do not need to use up additional memory space to store duplicate information for the copy. That is, we do not need to store the first and last names twice (i.e., for the original and the copy) since they are the same names. Normally this is not a problem when writing code as long as we know that the items are shared.

We need to understand the consequences of having such shared data. Consider what happens in the following example as we make changes to the copy:

Algorithm: CopyTest2

```
1.    original ← new array with capacity 3
2.    original[0] ← new Person("Hank", "Urchif", 21, 5.8, false)
3.    original[1] ← new Person("Holly", "Day", 32, 5.9, false)
4.    original[2] ← new Person("Bobby", "Pins", 87, 6.2, true)

5.    copy ← makeCopy(original)

6.    copy[0] .firstName ← "Steve"
7.    copy[2] ← new Person("April", "Rain", 15, 4.7, false)
8.    copy[2] .firstName ← "Acid"
```

We are doing three things to the copy: (1) changing a person's name, (2) replacing one person entirely with a new one, and (3) changing the new one's name. Here is what happened:



When changing Hank's name in the copy to "Steve", the original list is modified as well since the **Person** object, that both the original and copy lists were sharing, has now been altered. This can have serious consequences in your program since you may end up with a part of your code that unknowingly affects other parts of your program by modifying data structures that were not meant to be changed.

Some modifications, however, can be made to the copy that do not affect the original. For example, when replacing Bobby with the new person **April** in the **copy**, the original remains unchanged since we are simply moving a "pointer" in the copy to a new memory location. Then, when we replace April's name with "Acid", the original is not affected since nowhere does it refer to that newly create **Person** object.

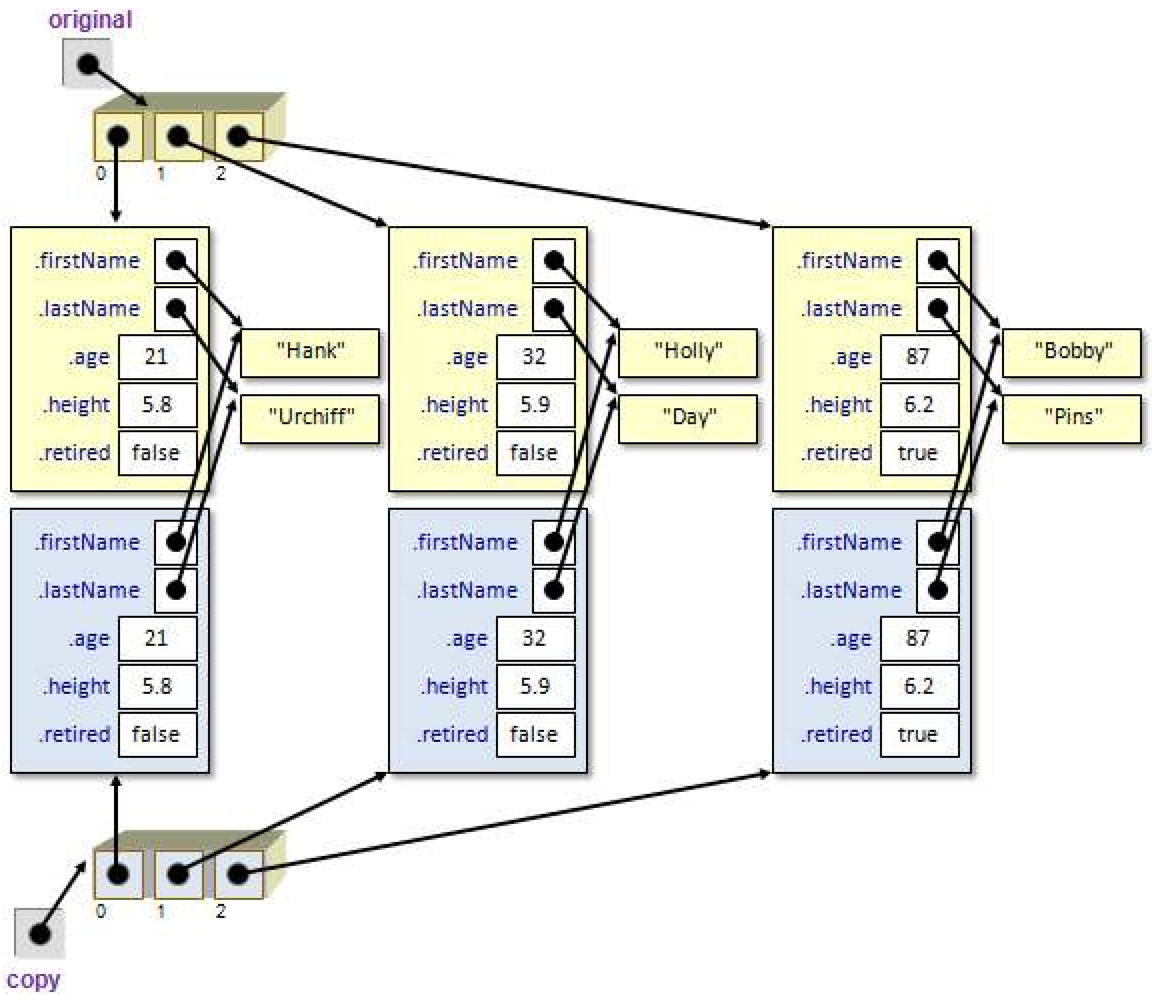
So you can see, by replacing, adding to or removing from a copied list, the original list remains intact. However, when we access a shared object from the copy and go into it to make changes to its attributes, then the original list will be affected.

To make a more thoroughly-separated copy, we could make what is known as a ***deep copy*** of the list so that the **Person** objects are not shared between them (i.e., each list has their own copies of the **Person** objects). To do this, we would need to make copies of the objects in the lists. That means, we would need to create a new object for each item in the list and then copy over all of the attributes so that they match the original objects.

Here is how we can modify the **makeCopy** function to accomplish this:

```
Function:  makeCopy(aList)

1.      anotherList ← new array with capacity aList.length
2.      for index i from 0 to aList.length - 1 {
3.          anotherList[i] ← new Person
4.          anotherList[i].firstName ← aList[i].firstName
5.          anotherList[i].lastName ← aList[i].lastName
6.          anotherList[i].age ← aList[i].age
7.          anotherList[i].height ← aList[i].height
8.          anotherList[i].retiredStatus ← aList[i].retiredStatus
        }
9.      return anotherList
```

Notice that much more work is involved. However, the effect of running our **CopyTest1** algorithm would be as follows:

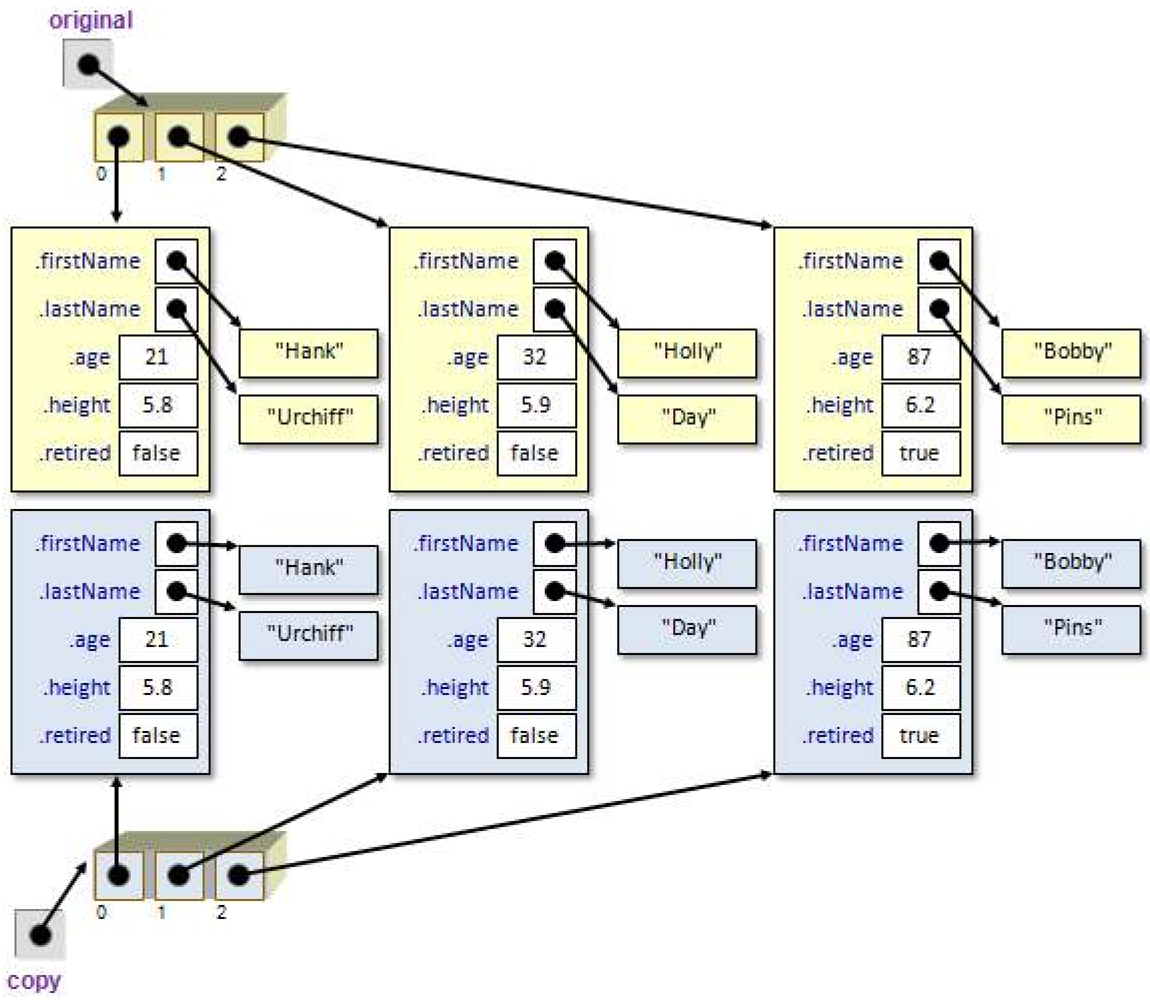


Notice that there are unique **Person** objects now and that changing the name of anyone in the copy will not affect the original. You may notice though, that the Strings are still shared! That means, if we altered any characters in one person's name in the copy, this would affect the original (i.e., remember...***replacing*** shared objects does not cause problems but ***modifying*** shared objects does cause a problem).

Fortunately, in Processing (and JAVA), it is not possible to modify characters in a String, so this would never be a problem. However, to be safe, a truly deep copy can be made by copying these strings as well by changing lines 4 and 5 in the **makeCopy** function to:

```
4.      anotherList[i].firstName ← new String with same characters as aList[i].firstName
5.      anotherList[i].lastName ← new String with same characters as aList[i].lastName
```

Then, we would have a true copy:



In fact, in order to make a truly deep copy, we would need to ensure that we ***thoroughly*** copy each of the attributes of all objects. That is, if an object is made up of other objects ... we must go into those other objects and make deep copies of them as well.

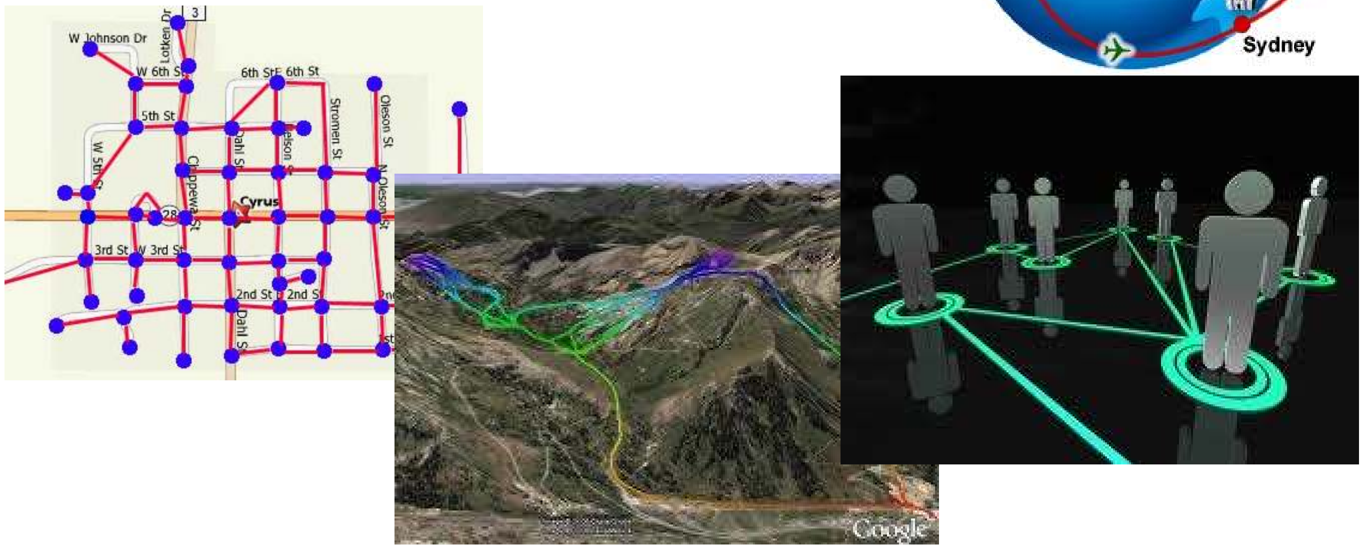
8.4 Example: Graph Editor

As a final example, we will consider an application in which data is shared within a data structure called a ***graph***:

> *A **graph** is an abstract data structure that represents interconnectivity of a set of objects (called **nodes** or Vertices). Two nodes of a graph are connected by an **edge** to indicate some kind of relationship between the nodes.*

Graphs are used throughout computer science. They can be used to store data that represents interconnectivity between various "things". For example, graphs can be used to represent

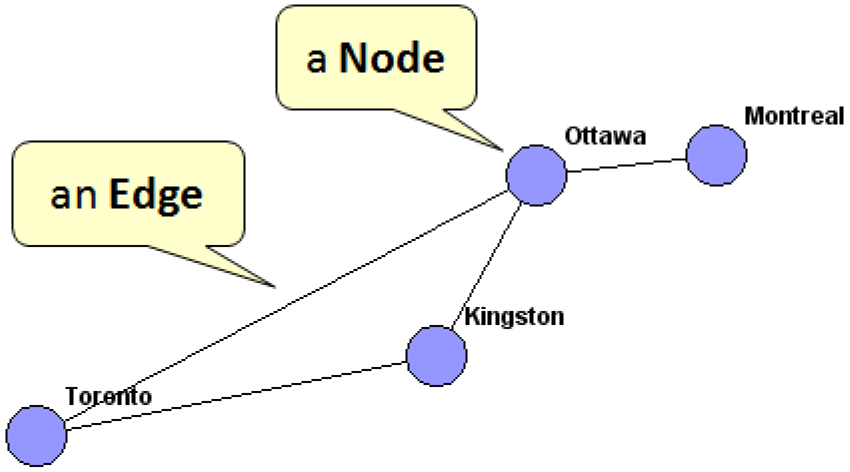
- connected flights from one city to another
- roads and intersections on a map
- various pathways on a terrain (e.g., water flow)
- interconnectivity in networks (e.g., social networks)
- etc...



Regardless of the application, graphs can be automatically generated or created manually. We will examine a simple application that allows the user to create, manipulate and edit a graph on a window in Processing.

This larger programming example with bring together many of the concepts that you have learned in the course including the use of data structures, graphics, event handling, arrays, searching and the sharing of data.

To begin, here is what the graph will look like:



We will then be able to move nodes and edges around the graph as well as add and delete nodes and edges. As you will see, the edges of the graph will share nodes so that when edges are moved, their nodes will also move and when a node is deleted, all edges connected to that node will be deleted.

We need to begin by understanding how the graph is stored. The graph will be made up of **Node** objects which have a particular (**x**,**y**) location on the screen as well as a **label**:

```
class Node {
  int        x, y;
  String     label;

  Node(int ix, int iy, String lab) {
    x = ix;
    y = iy;
    label = lab;
  }
}
```

Creating nodes is therefore quite easy, since we simply call the constructor as follows:

```
new Node(350, 250, "Ottawa");
```

Edges will connect two nodes. It does not make sense for an edge to exist unless it connects two nodes of the graph. For example, all of an airline's flights have a departure city and an destination city. It does not make sense for either one of these cities to be missing. Therefore, an edge will not connect two *points* on the screen, rather it will connect two nodes of the graph. Here is the definition of the **Edge** data structure:

```
class Edge {
  Node       startNode, endNode;

  Edge(Node start, Node end) {
    startNode = start;
    endNode = end;
  }
}
```

Notice how the attributes of the **Edge** are actually **Node** objects. Creating an edge therefore requires us to already have two **Node** objects:

```
Node n1 = new Node(350, 250, "Ottawa");
Node n2 = new Node(100, 380, "Toronto");

Edge e = new Edge(n1, n2);
```

Now, for the **Graph** itself ... it is really nothing other than a set of nodes and edges. We can store these using two arrays ... one for the nodes and one for the edges. But how big should the arrays be ? Ideally, we should allow them to grow (recall from chapter 5 how we were able to grow an array by making a new one and copying over the contents). However, to keep our code simple, we will not allow the arrays to grow but instead we will set a maximum number of nodes and edges that can be added to the graph. Here is our definition for the **Graph** data structure:

```
final int MAX_GRAPH_NODES = 100;       // arbitrarily chosen #
final int MAX_GRAPH_EDGES = 1000;      // arbitrarily chosen #

class Graph {
  int       numNodes, numEdges;
  Node[]    nodes;
  Edge[]    edges;

  Graph() {
    nodes = new Node[MAX_GRAPH_NODES];
    edges = new Edge[MAX_GRAPH_EDGES];
    numNodes = numEdges = 0;
  }
}
```

Graphs with no nodes or edges are made by simply calling the constructor: `new Graph();`

However, in order to make an interesting graph, we will need the ability to add nodes and edges to it. We can write simple procedures that allow us to add nodes and edges to the corresponding graph arrays as follows:

```
// Add the given node to the given graph
void addNode(Graph g, Node n) {
  if (g.numNodes < MAX_GRAPH_NODES) {
    g.nodes[g.numNodes] = n;
    g.numNodes++;
  }
}

// Add an edge to the given graph from node a to node b
void addEdge(Graph g, Node a, Node b) {
  if (g.numEdges < MAX_GRAPH_EDGES) {
    g.edges[g.numEdges] = new Edge(a, b);
    g.numEdges++;
  }
}
```

Notice the error checking to make sure that we do not go past the maximum array sizes.

Here is a simple function that creates and returns the graph shown earlier:

```
Graph exampleGraph() {
  Graph g = new Graph();
  Node n1 = new Node(350, 250, "Ottawa");
  Node n2 = new Node(100, 380, "Toronto");
  Node n3 = new Node(300, 340, "Kingston");
  Node n4 = new Node(440, 240, "Montreal");
  addNode(g, n1);
  addNode(g, n2);
  addNode(g, n3);
  addNode(g, n4);
  addEdge(g, n1, n2);
  addEdge(g, n1, n3);
  addEdge(g, n1, n4);
  addEdge(g, n2, n3);

  return g;
}
```

To display the graph, we need to write the **setup()** procedure that will define the window size, initialize the graph and prepare the font for the node labels:

```
Graph    graph;       // Global variable to store the graph

// Start the program by initializing everything
void setup() {
  size(600,600);
  graph = exampleGraph();

  // Set up the font for the node labels (must be in data directory)
  PFont font = loadFont("Arial-BoldMT-12.vlw");
  textFont(font);
}
```

The above code makes a 600x600 window and loads a font that will be used to draw the labels onto the nodes. It also creates the example graph and stores it in a global variable that we can access from anywhere within our program. (Note that the font string shown here was automatically produced from the **Tools/Create Font...** menu option in Processing).

Drawing a node **n** is as simple as calling the **ellipse()** procedure where the node's (x,y) position is its center and the NODE_DIAMETER is the width and height of the node:

```
final int NODE_RADIUS = 15;
final int NODE_DIAMETER = NODE_RADIUS*2;
ellipse(n.x, n.y, NODE_DIAMETER, NODE_DIAMETER);
```

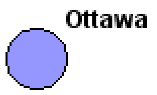
Drawing an edge **e** is also simple using the **line()** procedure:

```
line(e.startNode.x, e.startNode.y, e.endNode.x, e.endNode.y);
```

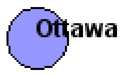
To draw the label of node **n**, we simply use the **text()** procedure:

```
    text(n.label, n.x + NODE_RADIUS, n.y - NODE_RADIUS);
```

Notice how we added the node's **radius** to the **x** and subtracted from the **y**. This allows us to draw the label up and to the right of the node ... making it more readable:



Drawing at (x+radius, y-radius)

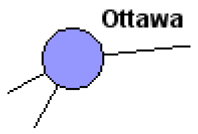


Drawing at (x,y)

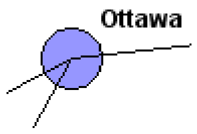
Here is the **draw()** procedure that draws the whole graph altogether:

```
    void draw() {
      background(255);    // Paint the background white
      stroke(0,0,0);      // black border

      // Draw the Edges
      for (int i=0; i<graph.numEdges; i++) {
        Edge e = graph.edges[i];
        line(e.startNode.x, e.startNode.y, e.endNode.x, e.endNode.y);
      }

      // Draw the Nodes
      for (int i=0; i<graph.numNodes; i++) {
        Node n = graph.nodes[i];
        fill(150,150,255);        // light blue inside color
        ellipse(n.x, n.y, NODE_DIAMETER, NODE_DIAMETER);
      }

      // Draw the labels on the nodes
      fill(0);
      for (int i=0; i<graph.numNodes; i++) {
        text(graph.nodes[i].label,
             graph.nodes[i].x + NODE_RADIUS,
             graph.nodes[i].y - NODE_RADIUS);
      }
    }
```

Notice that the code first clears the background, then draws the edges, then the nodes and finally the labels of the nodes. Drawing the edges first prevents the edges from drawing on top of the center of the nodes:

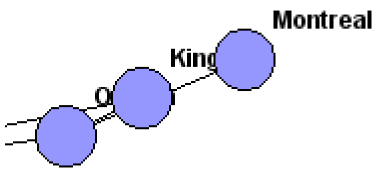


Edges drawn first

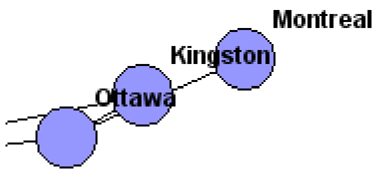


Nodes drawn first

Likewise, the labels of the nodes are drawn afterwards so that they will always appear on top and not get hidden underneath the other nodes:



Labels drawn with Nodes



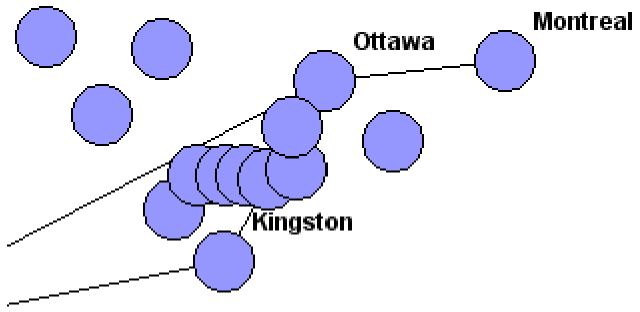
Labels drawn after nodes

Now that we have the ability to draw the graph, we can start working on allowing it to be edited and manipulated.

Adding Nodes:

To begin, we will write code that allows the user to add a node to the graph by double-clicking the left mouse button at some location in the window. The new node should appear centered where the user clicked. We need to write a **mouseClicked()** event handler:

```
void mouseClicked() {
  addNode(graph, new Node(mouseX, mouseY, ""));
}
```



Notice how simple the code is. We simply check if the mouse was clicked and add a node to the graph with the current mouse position as the node's center. We can now add new nodes to the graph ... as shown here →

To adjust the code so that the node only gets added when the mouse is double-clicked, we need to make use of a variable in Processing called **mouseEvent**. That variable is an object that allows us to apply various functions to it. In the next course, we will see that we can "attach" (or associate) functions and procedures to objects. In this case, we will call the **getClickCount()** function which returns the number of successive clicks that the user made so that we can add the node only when a 2nd successive click was made:

```
void mouseClicked() {
  if (mouseEvent.getClickCount() == 2)
    addNode(graph, new Node(mouseX, mouseY, ""));
}
```

Selecting Nodes:

Many drawing programs have the ability to select and move objects around. We will now add functionality to allow our nodes to be selected. Selected nodes should *appear* selected somehow ... in our case ... we will simply color them red instead of blue.

Rather than trying to keep a list of selected nodes, we can alter the **Node** data structure to include a **boolean** so that each node has the ability to be selected or unselected:

```
class Node {
  int        x, y;
  String     label;
  boolean    selected;

  Node(int ix, int iy, String lab) {
    x = ix;
    y = iy;
    label = lab;
    selected = false;
  }
}
```

So then what do we do to select a node ? Likely we will double-click on the node. What if we double-click on a node that is already selected ? It should become unselected. So, double-clicking on a node should cause it to toggle between being selected and unselected.

We need to modify our **mouseClicked()** event handler. There is a slight problem ... double-clicking anywhere on the window will cause a new node to be added. We need to adjust the code therefore so that when we double-click on top of a node ... we select it instead of adding a new node.

```
if (there is a node already at the mouse location)
     select the node
otherwise
     add a new node at the mouse location
```

So, we need to find out whether or not there is a node at the mouse location. We can write a function to find and return the node at a given location. To do this, we need to look at all the nodes and determine which one was clicked on (if any). Here is one way to do this:

```
Node nodeAt(Graph g, int x, int y) {
  for (int i=0; i<g.numNodes; i++) {
    if ((g.nodes[i].x == x) && (g.nodes[i].y == y))
      return g.nodes[i];
  }
  return null;
}
```

This code will look through all the nodes in the **graph** and find the one whose center matches the point specified by the given (**x,y**) point. If we find such a node, it is returned right away, otherwise the FOR loop completes ... no matching node was found ... so the function returns **null**.

The problem with this code, however, is that we are extremely unlikely to click on the exact center of the node. It would be very difficult and painfully tedious to force the user to click on the center of a node to select it. So, we need to adjust our code so that the user may click anywhere inside the node to select it. How do we determine whether the point (**x,y**) is inside the node or outside of it ? We already solved this problem when we simulated the throwing of a ball back in chapter 3. We just need to check the distance from the click point to the center of the node and ensure that it is less than the radius of the node:

```
        Node nodeAt(Graph g, int x, int y) {
          for (int i=0; i<g.numNodes; i++) {
            Node n = g.nodes[i];
            float d = sqrt(((n.x - x) * (n.x - x)) + ((n.y - y) * (n.y - y)));
            if (d <= NODE_RADIUS)
              return n;
          }
          return null;
        }
```

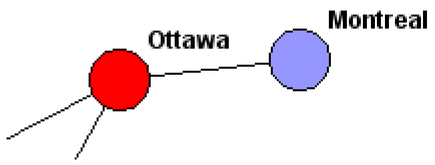
Note that the code above can be made more efficient by checking the "square" of the distance against the square of the radius to avoid the computation of a square root ... thereby speeding up the computation.

Now we can adjust the **mouseClicked()** event handler to allow nodes to be selected:

```
        void mouseClicked() {
          if (mouseEvent.getClickCount() == 2) {
            Node n = nodeAt(graph, mouseX, mouseY);
            if (n != null)
              n.selected = !n.selected;
            else
              addNode(graph, new Node(mouseX, mouseY, ""));
          }
        }
```

All that remains to be done is to draw nodes as selected. We can replace the `fill(150, 150, 255)` line in the **draw()** procedure by the following:

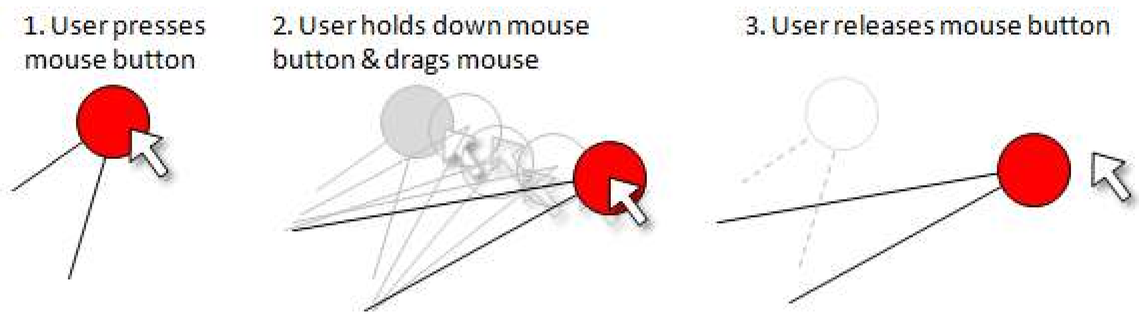
```
      if (n.selected)
        fill(255,0,0);      // red inside
      else
        fill(150,150,255); // light blue inside
```



We can now select nodes.

Moving Nodes:

A natural user action to move nodes around on the window would be to select a node and then grab it and drag it to its new position. To allow this functionality, we will need to incorporate into our program additional event handlers. Let us consider what happens when the user drags a node:


1. User presses mouse button
2. User holds down mouse button & drags mouse
3. User releases mouse button

So, we will need to write **mousePressed**, **mouseDragged** and **mouseReleased** event handlers. What happens when the mouse is pressed on a selected node ? Well, the node does not move. Nothing really "happens" visually. What about when the mouse is dragged ? Well, we need to move the node that we had clicked on to the new mouse location. Within the **mouseDragged** event handler, we will need to know which node we had initially pressed the mouse on. So, in the **mousePressed()** event handler all we need to do is to find out which node was at that position and remember it. We can store this into a variable called **dragNode** as follows:

```
Node     dragNode;    // Add to the top of program

void mousePressed() {
  dragNode = nodeAt(graph, mouseX, mouseY);
}
```

Then, as the mouse is dragged, we simply access this **dragNode** and set its (x,y) location to be the mouse's location as follows:

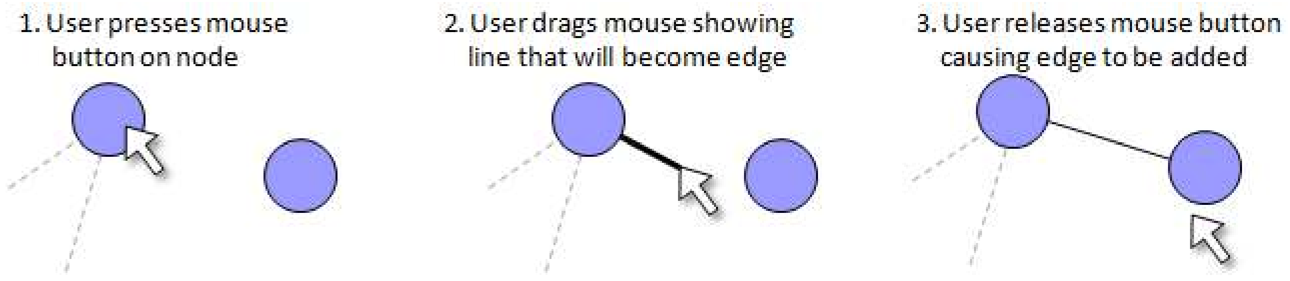
```
void mouseDragged() {
  dragNode.x = mouseX;
  dragNode.y = mouseY;
}
```

However, it is possible that the user may drag the mouse without first having pressed on a node. In that case, the **nodeAt()** function will return **null** (as it is supposed to) and our code would stop working since **dragNode** would be **null** and we cannot access **null.x** or **null.y**. So we should check to ensure that **dragNode** is not **null**. Also, we should ensure that the **dragNode** is **selected**, otherwise we may be moving nodes that we did not intend to move. Here are the changes:

```
void mouseDragged() {
  if (dragNode != null) {
    if (dragNode.selected) {
      dragNode.x = mouseX;
      dragNode.y = mouseY;
    }
  }
}
```

Adding Edges:

What kind of user-action will our program consider for adding edges ? There are a few options. For example, we could write code that allows an edge to be added simply by clicking two nodes one after another. However, a "nicer" interface will allow the user to click on a node and "drag" a line towards another edge so that when the user lets go on another node, an edge is added:


1. User presses mouse button on node
2. User drags mouse showing line that will become edge
3. User releases mouse button causing edge to be added

The idea is similar to that of "stretching" an elastic band tied from the center of the start node with the other end on the mouse cursor. This ***elastic-banding*** effect is common with many drawing programs as it provides feedback to the user giving a rough idea as to how the edge is going to appear in the graph.

To accomplish this, as the mouse is being dragged, we will need to draw a line from the start node to the mouse's current location. Where do we do the drawing of this line ? It needs to be done in the **draw()** procedure. So, in the **draw()** procedure we will need to know the start node as well as the current mouse location.

The code in the **mousePressed** event handler already stored the start node as **dragNode**. All we would then need to do is to ensure that as the user drags the mouse (i.e., in the **mouseDragged** event handler) we store the mouse location. We could use variables called **elasticEndX** and **elasticEndY** as follows:

```
int     elasticEndX, elasticEndY;     // Add to the top of program

void mouseDragged() {
  if (dragNode != null) {
    if (dragNode.selected) {
      dragNode.x = mouseX;
      dragNode.y = mouseY;
    }
    else { // store end point for edge to be added
      elasticEndX = mouseX;
      elasticEndY = mouseY;
    }
  }
}
```

Then, in the **draw()** procedure we add code to ensure that the line is drawn. However, we want to make sure that the user first clicked on a node to begin the edge adding process.

If the user clicked off of a node, then we can detect this by checking if **dragNode** is **null** as follows:

```
// Draw the elastic band
stroke(0,0,0); //black
if (dragNode != null)
 line(dragNode.x, dragNode.y, elasticEndX, elasticEndY);
```

This code should appear after **background(255)** but before the nodes of the graph are drawn so that the line will appear behind the nodes and not cut across them.

If the user decides to start drawing an edge but then lets go of the mouse button somewhere other than on another node, then the edge should not be added and the line should disappear. However, our code does not do this. We need to have a way of informing the **draw()** procedure NOT to draw the line once the user releases the mouse button. The simplest way to do this is to set the **elasticEndX** to have a value of **-1**... or something similar ... in the **mouseReleased** event handler. Then we can check for this in the **draw()** procedure:

```
void mouseReleased() {
  elasticEndX = -1;
  elasticEndY = -1;
}

void draw() {
  ...
  if ((dragNode != null) && (elasticEndX != -1))
    line(dragNode.x, dragNode.y, elasticEndX, elasticEndY);
  ...
}
```

Of course, we must handle the case where the user successfully releases the mouse on a node. Then we need to add a new edge. We should add the edge as long as the user released the mouse on a "different" node...not the same one as the dragNode:

```
void mouseReleased() {
  elasticEndX = -1;
  elasticEndY = -1;
  if (dragNode != null) {
    Node n = nodeAt(graph, mouseX, mouseY);
    if ((n != null) && (n != dragNode))
      addEdge(graph, dragNode, n);
    }
  }
}
```

We now have the ability to add nodes to the graph. You may notice that dragging a "selected" node does not add an edge but still allows the user to move the node around.

Deleting Nodes:

Of course, sooner or later we are going to create some nodes by mistake and want to delete them. With many drawing programs, a common action for deleting objects is to first select them and then press delete.

How do we allow them to be deleted ? We allow the user to press the **DELETE** key. This should cause them to be deleted. We simply write a **keyPressed()** event handler and determine if the **key** variable has been set to **DELETE**. Then we delete any nodes that have been selected:

```
void keyPressed() {
  if (key == DELETE) {
    for (int i=0; i<graph.numNodes; i++) {
      if (graph.nodes[i].selected) {
        deleteNode(graph, graph.nodes[i]);
        i--; // required so that we check the same index next time
      }
    }
  }
}
```

Notice that that the code loops through all nodes, determines if they are selected and then calls a **deleteNode()** procedure (we need to write this yet). The **i--** will make sense after we write the **deleteNode()** procedure.

Deleting a node is actually a complicated process since there may be edges connected to a node. We cannot allow any edges to remain in the graph if the edge's **startNode** or **endNode** has been deleted. So we first need to delete all the edges connected to the node being deleted, then we can delete the node itself. We can write a procedure that loops through the edges and deletes the ones that have the given node as its start or end node as follows:

```
void deleteNode(Graph g, Node n) {
  // Find & delete the edges connected to this node
  for (int j=0; j<g.numEdges; j++) {
    if ((g.edges[j].startNode == n) || (g.edges[j].endNode == n)) {
      // Put the last edge in the place of this deleted edge
      g.edges[j] = g.edges[g.numEdges-1];
      g.numEdges--;
      j--;   // required so that we check the same index next time
    }
  }
  // Now find and delete the Node
  for (int i=0; i<g.numNodes; i++) {
    if (g.nodes[i] == n) {
      // Put the last node in the place of this deleted node
      g.nodes[i] = g.nodes[g.numNodes-1];
      g.numNodes--;
      return;
    }
  }
}
```

Notice that to delete something from an array we need to replace it with a different value. Rather than leave gaps, the above code takes the last edge (or node ... depending on the array) and places it in the array at the position of the one deleted. Then the number of elements in the array is actually smaller, so we decrease **numEdges** (or **numNodes** ... depending on the array). The **j--** is required to ensure that the next time through the loop ... we check this same location that we removed the edge (or node) from ... since we placed the last edge (or node) there and it too needs to be checked.

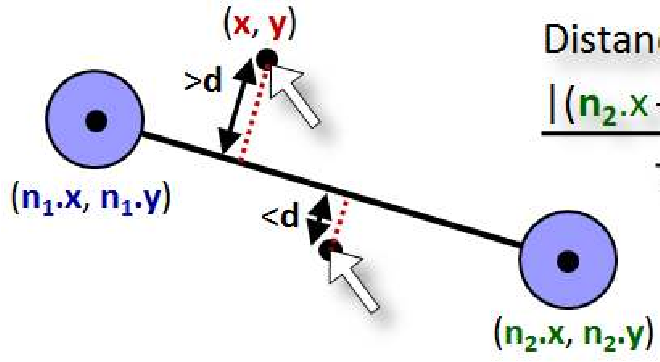
The code should now work, allowing all selected nodes to be deleted.

Selecting Edges:

We will also want to select, move and delete edges. We can add a **boolean** to the edge data structure to allow this:

```
class Edge {
  Node       startNode, endNode;
  boolean    selected;

  Edge(Node start, Node end) {
    startNode = start;
    endNode = end;
    selected = false;
  }
}
```

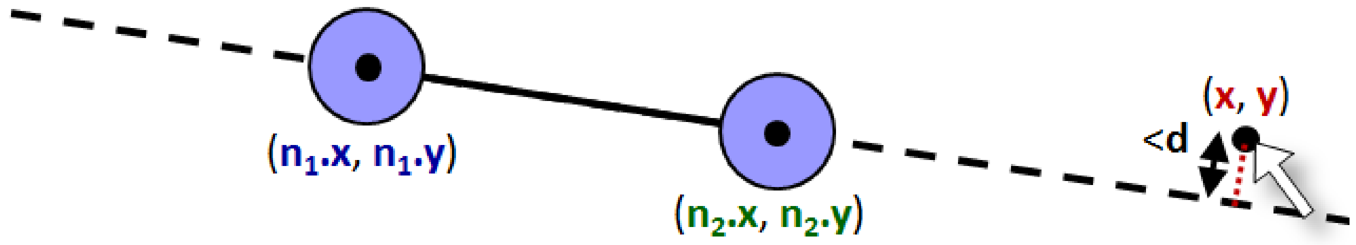
How does the user select an edge ? Likely, by clicking on or near it. We can accomplish this by determining the distance between the point clicked at and the edge itself. If the distance is smaller than some pre-decided value (e.g., 5 pixels) then we can assume that this edge was just clicked on... otherwise 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:



Distance from (x, y) to edge (n_1, n_2) is:

$$\frac{|(n_2.x - n_1.x)(n_1.y - y) - (n_1.x - x)(n_2.y - n_1.y)|}{\sqrt{(n_2.x - n_1.x)^2 + (n_2.y - n_1.y)^2}}$$

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 that 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
}
```

Now that we have the algorithm that we need, it would make sense to create an **edgeAt()** function to find the edge that was selected, since we did something similar for **nodeAt()**.

Here is the code:

```
Edge edgeAt(Graph g, int x, int y) {
  for (int i=0; i<g.numEdges; i++) {
    Edge e = g.edges[i];
    Node n1 = e.startNode;
    Node n2 = e.endNode;
    int xDiff = n2.x - n1.x;
    int yDiff = n2.y - n1.y;
    float distance = abs(xDiff*(n1.y-y)-(n1.x-x)*yDiff) /
                     sqrt(xDiff*xDiff + yDiff*yDiff);
    if (distance <= 5) {
      if (abs(xDiff) > abs(yDiff)) {
        if (((x < n1.x) && (x > n2.x)) || ((x > n1.x) && (x < n2.x)))
          return e;
      }
      else
        if (((y < n1.y) && (y > n2.y)) || ((y > n1.y) && (y < n2.y)))
          return e;
    }
  }
  return null;
}
```

Now that we have such a function, we can modify the **mouseClicked** event handler to allow edges to be selected:

```
void mouseClicked() {
  if (mouseEvent.getClickCount() == 2) {
    Node n = nodeAt(graph, mouseX, mouseY);
    if (n != null)
      n.selected = !n.selected;
    else {
      Edge e = edgeAt(graph, mouseX, mouseY);
      if (e != null)
        e.selected = !e.selected;
      else
        addNode(graph, new Node(mouseX, mouseY, ""));
    }
  }
}
```

Notice now that we first check to see if a node was selected and only if a node was not selected will we then check to see if an edge was selected. Of course, we will want to show selected edges as red, so we need to modify the **draw()** procedure:

```
if (e.selected) {
  stroke(255,0,0);  // red edge
  strokeWeight(5);  // thicker line
}
else {
  stroke(0,0,0);    // black edge
  strokeWeight(1);  // thin line
}
```

Our program now allows edges to be selected.

Deleting Edges:

We will want to allow edges to be deleted as well. To do this, we should allow the user to select any edges and press delete. To delete an edge, we simply need to remove it from the array. We can write a **deleteEdge()** procedure to do this as follows:

```
void deleteEdge(Graph g, Edge e) {
  for (int i=0; i<g.numEdges; i++) {
    if (g.edges[i] == e) {
      // Put the last edge in the place of this deleted edge
      g.edges[i] = g.edges[g.numEdges-1];
      g.numEdges--;
      return;
    }
  }
}
```

Then, in the **keyPressed** event handler we simply remove any selected edges by adding the following code:

```
for (int i=0; i<graph.numEdges; i++) {
  if (graph.edges[i].selected) {
    deleteEdge(graph, graph.edges[i]);
    i--;
  }
}
```

Moving Edges:

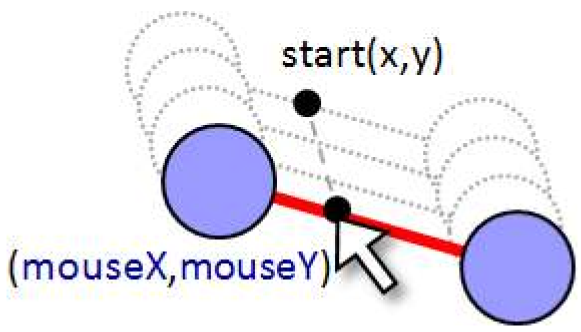
It would be nice to allow an edge to be grabbed and dragged just like a node is grabbed and dragged. To do this, we will need to determine whether or not the user has pressed the mouse on an edge and then "remember" that edge so that it can be dragged later. For nodes, we used a **dragNode** variable. For edges we can create a similar **dragEdge** variable:

```
Edge     dragEdge;
```

We can set this to **null** in the **setup()** procedure and then in the **mousePressed** event handler we can set it as follows:

```
dragEdge = edgeAt(graph, mouseX, mouseY);
```

At this point, we will know which edge has been selected for dragging. All that is left to do is to actually move the edge when the user drags the mouse. However, how do we actually move the edge ? It should move by the amount that the mouse moved. We can calculate the amount that the mouse moved by remembering the "start" point that was initially clicked on and then finding the change in **x** and **y** to the current mouse location:



So, in addition to the edge that has been selected, we will need to remember the point that was clicked at so that we can determine the change from that point to the new mouse location in order to adjust the edge's node locations. We can create two new variables at the top of the program to store the mouse location at which the user initially pressed:

```
int      dragX, dragY;
```

We can then add this to the **mousePressed** handler set the values:

```
dragX = mouseX;
dragY = mouseY;
```

Then in the **mouseDragged** event handler we can now move the edge, provided that one was selected by the user by appending this code:

```
if (dragEdge != null)
    if (dragEdge.selected) {
      int x = mouseX - dragX;
      int y = mouseY - dragY;
      dragEdge.startNode.x = dragEdge.startNode.x + x;
      dragEdge.startNode.y = dragEdge.startNode.y + y;
      dragEdge.endNode.x = dragEdge.endNode.x + x;
      dragEdge.endNode.y = dragEdge.endNode.y + y;
      dragX = mouseX;
      dragY = mouseY;
    }
```

Moving Multiple Nodes:

When we moved the edge in the above code, we actually did this by moving two nodes. In fact, we can adjust our node-dragging code so that it allows multiple selected nodes to be moved around at the same time. We just need to offset the nodes by an amount equal to the difference between the current mouse location (**mouseX**, **mouseY**) and the (**dragX**, **dragY**) location ... as when we moved the edge.

So, instead of doing this in the **mouseDragged** event handler:

```
dragNode.x = mouseX;  dragNode.y = mouseY;
```

we can do the following to move ALL the selected nodes:

```
    for(int i=0; i<graph.numNodes; i++) {
      if (graph.nodes[i].selected) {
        graph.nodes[i].x = graph.nodes[i].x + x;
        graph.nodes[i].y = graph.nodes[i].y + y;
      }
    }
```

Now, for example, if we selected all the nodes in the graph, we could translate the whole graph with one drag operation !

Other Fun Features:

If you want to have more fun, try implementing these features on your own:

- 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.