Syntax for a standalone application in Java:

```
class <classname>
{
     public static void main(String args[])
      {
           statements;
           ________;
           ________;
      }
}
```

Steps to run the above application:

1. Type the program in the DOS editor or notepad. Save the file with a .java extension.
2. The file name should be the same as the class, which has the main method.
3. To compile the program, using javac compiler, type the following on the command line:
   Syntax: `javac <filename.java>`
   Example: `javac abc.java`
4. After compilation, run the program using the Java interpreter.
   Syntax: `java <filaname>` (without the .java extension)
   Example: `java abc`
5. The program output will be displayed on the command line.

Java reserved words:

abstract	default	if	package	this
boolean	do	implements	private	throw
Break	double	import	protected	throws
Byte	else	instanceof	public	transient
case	extends	int	return	null
try	Const	for	new	switch
continue	while	goto	synchronized	super
Catch	final	interface	short	void
char	finally	long	static	volatile
class	float	native		

Java naming conventions:

Variable Names: Can start with a letter, '$' (dollar symbol), or '_' (underscore); cannot start with a number; cannot be a reserved word.

Method Names: Verbs or verb phrases with first letter in lowercase, and the first letter of subsequent words capitalized; cannot be reserved words.
Example: `setColor()`

Class And Interface Names: Descriptive names that begin with a capital letter, by convention; cannot be a reserved word.

Constant Names: They are in capitals.
Example: `Font.BOLD, Font.ITALIC`

Java Comments:

Delimiters	Use
//	Used for commenting a single line
/* ———— */	Used for commenting a block of code
/** ————*/	Used for commenting a block of code. Used by the Javadoc tool for generating Java documentation.

Primitive datatypes in Java:

DataType	Size	Default	Min Value Max Value
byte (Signed integer)	8 bits	0	-128 +127
short (Signed integer)	16 bits	0	-32,768 +32,767
int (Signed integer)	32 bits	0	-2,147,483,648 +2,147,483,647
long (Signed Integer)	64 bits	0	-9, 223, 372,036,854, 775,808, +9,223,372,036, 854, 775, 807
float (IEEE 754 floating-point)	32 bits	0.0	1.4E-45 3.4028235E38
double (IEEE 754 floating-point)	64 bits	0.0	4.9E-324 1.7976931348623157E308
char (Unicode character)	16 bits	\u0000	\u0000 \uFFFF
boolean	1 bit	false	

Variable Declaration:
`<datatype>  <variable name>`
Example: `int num1;`

Variable Initialization:
`<datatype>   <variable name> = value`
Example: `double num2 = 3.1419;`

Escape sequences:

Literal	Represents
\n	New line
\t	Horizontal tab
\b	Backspace
\r	Carriage return

\f	Form feed
\\	Backslash
\”	Double quote
\ddd	Octal character
\xdd	Hexadecimal character
\udddd	Unicode character

Arrays: An array which can be of any datatype, is created in two steps – array declaration and memory allocation.

Array declaration

```
<datatype> []  <arr```````````ayname>;
```

Examples

```
int[]  myarray1;
double[] myarray2;
```

Memory Allocation

The new keyword allocates memory for an array.

Syntax

```
<arrayname> = new <array type> [<number of
elements>];
```

Examples

```
myarray1 = new int[10];
Myarray2 = new double[15];
```

Multi-dimensional arrays:

Syntax:

```
<datatype> <arrayname> [] [] = new <datatype>
[number of rows][number of columns];
```

Example:

```
int mdarray[][] = new int[4][5];
```

Flow Control:

1. If........else statements

Syntax:

```
 if(condition)
{
  statements;
}
else
{
 statements;
}
```

2. For loop

Syntax:

```
 for(initialization; condition; increment)
{
  statements;
 }
```

3. While loop

Syntax:

```
while(condition)
{
  statements;
}
```

4. Do....While loop

Syntax:

```
do
{
    statements;
}
while(condition);
```

5. Switch statement

Syntax:

```
 switch(variable)
{
  case(value1):
  statements;
  break;
  case(value2):
  statements;
  break;
  default:
  statements;
  break;
  }
```

Class Declaration: A class must be declared using the keyword `class` followed by the class name.

Syntax

```
class <classname>
{
  ——— Body of the class
```

A typical class declaration is as follows:

```
<modifier> class <classname> extends
<superclass name> implements <interface name>
{
  ———Member variable declarations;
  ———Method declarations and definitions
}
```

Member variable declarations:

```
<access specifier> <static/final/transient/
volatile> <datatype> <variable name>
```

Example `public final int num1;`

Method declarations:

```
<access specifier> <static/final> <return type>
<method name> <arguments list>
{
  Method body;
}
```

Example

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

Interface declaration: Create an interface. Save the file with a.java extension, and with the same name as the interface. Interface methods do not have any implementation and are abstract by default.

Syntax

```
 interface <interface name>
{
  void abc();
  void xyz();
}
```

Using an interface: A class implements an interface with the `implements` keyword.

Syntax
```
class <classname> extends <superclass name>
implements <interface name>
{
  class body;
  ————————;
}
```

Creating A Package:

1. Identify the hierarchy in which the `.class` files have to be organized.
2. Create a directory corresponding to every package, with names similar to the packages.
3. Include the package statement as the first statement in the program.
4. Declare the various classes.
5. Save the file with a .java extension.
6. Compile the program which will create a .class file in the same directory.
7. Execute the .class file.

Packages and Access Protection:

Accessed	**Public**	**Protected**	**Package**	**Private**
From the same class ?	Yes	Yes	Yes	Yes
From a non subclass in the same package ?	Yes	Yes	Yes	No
From a non subclass outside the package?	Yes	No	No	No
From a subclass in the same package?	Yes	Yes	Yes	No
From a subclass outside the package ?	Yes	Yes	No	No

Attribute modifiers in Java:

Modifier	**Acts on**	**Description**
abstract	Class	Contains abstract methods.Cannot be instantiated.
	Interface	All interfaces are implicitly abstract. The modifier is optional.
	Method	Method without a body. Signature is followed by a semicolon. The class must also be abstract.
final	Class	Cannot be subclassed.
	Method	Cannot be overridden.
	Variable	Value cannot be changed (Constant)
native	Method	Implemented in a language other than Java like C,C++, assembly etc. Methods do not have bodies.
static	Method	Class method. It cannot refer to nonstatic variables and methods of the class. Static methods are implicitly final and invoked through the class name.
	Variable	Class variable. It has only one copy regardless of how many instances are created. Accessed only through the class name.
synchronized	Method	A class which has a synchronized method automatically acts as a lock. Only one synchronized method can run for each class.

List of exceptions in Java(part of java.lang package):

Essential exception classes include -

Exception	**Description**
`ArithmeticException`	Caused by exceptional conditions like divide by `zero`
`ArrayIndexOfBounds Exception`	Thrown when an array is accessed beyond its bounds
`ArrayStoreException`	Thrown when an incompatible type is stored in an array
`ClassCastException`	Thrown when there is an invalid cast
`IllegalArgument Exception`	Thrown when an inappropriate argument is passed to a method
`IllegalMonitorState Exception`	Illegal monitor operations such as waiting on an unlocked thread
`IllegalThreadState Exception`	Thrown when a requested operation is incompatible with the current thread state.
`IndexOutOfBounds Exception`	Thrown to indicate that an index is out of range.
`NegativeArraySize Exception`	Thrown when an array is created with negative size.

`NullPointerException`	Invalid use of a null reference.
`NumberFormatException`	Invalid conversion of a string to a number.
`SecurityException`	Thrown when security is violated.
`ClassNotFound Exception`	Thrown when a class is not found.
`CloneNotSupported Exception`	Attempt to clone an object that does not implement the Cloneable interface.
`IllegalAccess Exception`	Thrown when a method does not have access to a class.
`Instantiation Exception`	Thrown when an attempt is made to instantiate an abstract class or an interface.
`InterruptedException`	Thrown when a second thread interrupts a waiting, sleeping, or paused thread.

The java.lang.Thread class

The Thread class creates individual threads. To create a thread either (i) extend the Thread class or (ii) implement the Runnable interface. In both cases, the `run()` method defines operations

performed by the thread.

Methods of the Thread class:

Methods	**Description**
`run()`	Must be overridden by `Runnable` object; contains code that the thread should perform
`start()`	Causes the `run` method to execute and start the thread
`sleep()`	Causes the currently executing thread to wait for a specified time before allowing other threads to execute
`interrupt()`	Interrupts the current thread
`Yield()`	Yields the CPU to other runnable threads
`getName()`	Returns the current thread's name
`getPriority()`	Returns the thread's priority as an integer
`isAlive()`	Tests if the thread is alive; returns a Boolean value
`join()`	Waits for specified number of milliseconds for a thread to die
`setName()`	Changes the name of the thread

`setPriority()`	Changes the priority of the thread
`currentThread()`	Returns a reference to the currently executing thread
`activeCount()`	Returns the number of active threads in a thread group

Exception Handling Syntax:

```
try
{
   //code to be tried for errors
}
 catch(ExceptionType1 obj1)
{
   //Exception handler for ExceptionType1
}
  catch(ExceptionType2 obj2)
{
   //Exception handler for ExceptionType2
}
 finally{
 //code to be executed before try block ends.
    This executes whether or not an //
    exception occurs in the try block.
 }
```

I/O classes in Java (part of the java.io package):

I/O class name	**Description**
`BufferedInputStream`	Provides the ability to buffer the input. Supports `mark()` and `reset()` methods.
BufferedOutputStream	Provides the ability to write bytes to the underlying output stream without making a call to the underlying system.
BufferedReader	Reads text from a character input stream
BufferedWriter	Writes text to character output stream
DataInputStream	Allows an application to read primitive datatypes from an underlying input stream
DataOutputStream	Allows an application to write primitive datatypes to an output stream
File	Represents disk files and directories
FileInputStream	Reads bytes from a file in a file system
FileOutputStream	Writes bytes to a file
ObjectInputStream	Reads bytes i.e. deserializes objects using the `readObject()` method
ObjectOutputStream	Writes bytes i.e. serializes objects using the `writeObject()`method
PrintStream	Provides the ability to print different data values in an efficient manner
RandomAccessFile	Supports reading and writing to a random access file

StringReader	Character stream that reads from a string
StringWriter	Character stream that writes to a `StringBuffer` that is later converted to a String

The java.io.InputStream class: The `InputStream` class is at the top of the input stream hierarchy. This is an abstract class which cannot be instantiated. Hence, subclasses like the `DataInputStream` class are used for input purposes.

Methods of the InputStream class:

Method	Description
available()	Returns the number of bytes that can be read
close()	Closes the input stream and releases associated system resources
mark()	Marks the current position in the input stream
mark Supported()	Returns true if mark() and reset() methods are supported by the input stream
read()	Abstract method which reads the next byte of data from the input stream
read(byte b[])	Reads bytes from the input stream and stores them in the buffer array

skip()	Skips a specified number of bytes from the input stream

The java.io.OutputStream class: The `OutputStream` class which is at the top of the output stream hierarchy, is also an abstract class, which cannot be instantiated. Hence, subclasses like `DataOutputStream` and `PrintStream` are used for output purposes.

Methods of the OutputStream class:

Method	Description
close()	Closes the output stream, and releases associated system resources
write(int b)	Writes a byte to the output stream
write(byte b[])	Writes bytes from the byte array to the output stream
flush()	Flushes the ouput stream, and writes buffered output bytes

java.io.File class: The `File` class abstracts information about files and directories.

Methods of the File class:

Method	Description
exists()	Checks whether a specified file exists
getName()	Returns the name of the file and directory denoted by the path name
isDirectory()	Tests whether the file represented by the pathname is a directory
lastModified()	Returns the time when the file was last modified
length()	Returns the length of the file represented by the pathname
listFiles()	Returns an array of files in the directory represented by the pathname
setReadOnly()	Marks the file or directory so that only read operations can be performed
renameTo()	Renames the file represented by the pathname
delete()	Deletes the file or directory represented by the pathname
canRead()	Checks whether the application can read from the specified file
canWrite()	Checks whether an application can write to a specified file

Creating applets:

1. Write the source code and save it with a .java extension
2. Compile the program
3. Create an HTML file and embed the `.class` file with the `<applet>` tag into it.
4. To execute the applet, open the HTML file in the browser or use the appletviewer utility, whch is part of the Java Development Kit.

The <applet> tag: `Code, width,` and `height` are mandatory attributes of the `<applet>` tag. Optional attributes include `codebase, alt,name, align, vspace,` and `hspace`. The code attribute takes the name of the class file as its value.
Syntax:

```
<applet code = “abc.class” height=300
width=300>
<param name=parameterName1   value= value1 >
<param name=parameterName2   value= value2 >
</applet>
```

Using the Appletviewer: `Appletviewer.exe` is an application found in the BIN folder as part of the JDK. Once an HTML file containing the class file is created (eg. `abc.html`), type in the command line:

```
Appletviewer    abc.html
```

java.applet.Applet class:

Methods of the java.applet.Applet class:

Method	Description
init()	Invoked by the browser or the applet viewer to inform that the applet has been loaded
start()	Invoked by the browser or the applet viewer to inform that applet execution has started
stop()	Invoked by the browser or the applet viewer to inform that applet execution has stopped

`destroy()`	Invoked by the browser or the appletviewer to inform that the applet has been reclaimed by the Garbage Collector
`getAppletContext()`	Determines the applet context or the environment in which it runs
`getImage()`	Returns an Image object that can be drawn on the applet window
`getDocumentBase()`	Returns the URL of the HTML page that loads the applet
`getCodeBase()`	Returns the URL of the applet's class file
`getParameter()`	Returns the value of a named applet parameter as a string
`showStatus()`	Displays the argument string on the applet's status

java.awt.Graphics class: The `Graphics` class is an abstract class that contains all the essential drawing methods like `drawLine()`, `drawOval()`, `drawRect()` and so on. A `Graphics` reference is passed as an argument to the `paint()` method that belongs to the `java.awt.Component` class.

Methods of the `Graphics` class:

Method	**Description**
`drawLine()`	Draws a line between (x1,y1) and (x2,y2) passed as parameters
`drawRect()/fillRect()`	Draws a rectangle of specified width and height at a specified location
`drawOval()/fillOval()`	Draws a circle or an ellipse that fills within a rectangle of specified coordinates
`drawString()`	Draws the text given as a specified string
`drawImage()`	Draws the specified image onto the screen

`drawPolygon()`
`/fillPolygon()`
	Draws a closed polygon defined by arrays of x and y coordinates

`setColor()`	Sets the specified color of the graphics context
`setFont()`	Sets the specified font of the graphics context

java.awt.Component class: The `Component` class is an abstract class that is a superclass of all AWT components. A component has a graphical representation that a user can interact with. For instance, `Button`, `Checkbox`, `TextField`, and `TextArea`.

Methods of the `Component` class:

Method	**Description**
`paint(Graphics g)`	Paints the component. The Graphics context g is used for painting.
`setBackground()`	Sets the background color of the component
`setForeground()`	Sets the foreground color of the component
`SetSize()`	Resizes the component
`setLocation()`	Moves the component to a new location
`setBounds()`	Moves the component to specified location and resizes it to the specified size
`addFocusListener()`	Registers a `FocusListener` object to receive focus events from the component
`addMouseListener()`	Registers a `MouseListener` object to receive mouse events from the component
`addKeyListener()`	Registers a `KeyListener` object to receive key events from the component
`getGraphics()`	Returns the graphics context of this component
`update(Graphics g)`	Updates the component. Calls the `paint()` method to redraw the component.

AWT Components: Many AWT classes like `Button`, `Checkbox`, `Label`, `TextField` etc. are subclasses of the `java.awt.Component` class. Containers like `Frame` and `Panel` are also subclasses of components, but can additionally hold other components.

Label:

Constructors

- `Label()` - Creates an empty label
- `Label(String s)` - Creates a label with left justified text string
- `Label (String s, int alignment)` - Creates a label with the specified text and specified aligment. Possible values for alignment could be `Label.RIGHT`, `Label.LEFT`, or `Label.CENTER`

Methods of the Label class:

Method	**Description**
`getAlignment()`	Returns an integer representing the current alignment of the Label. 0 for left, 1 for center, and 2 for right alignment.
`setAlignment()`	Sets the alignment of the Label to the specified one
`getText()`	Returns the label's text as a string
`setText()`	Sets the label's text with the specified string

Button:

Constructors

`Button()` - Creates a button without a label
`Button(String s)` - Creates a button with the specified label

Methods of the `Button` class:

Method	Description
`addActionListener()`	Registers an `ActionListener` object to receive action events from the button
`getActionCommand()`	Returns the command name of the action event fired by the button. Returns the button label if the command name is null.
`GetLabel()`	Returns the button's label
`SetLabel()`	Sets the button's label to the specified string

Checkbox:

Constructors

- `Checkbox()` - Creates a checkbox without any label
- `Checkbox(String s)` - Creates a checkbox with a specified label
- `Checkbox(String s, boolean state)` - Creates a checkbox with a specified label, and sets the specified state
- `Checkbox(String s, boolean state, CheckboxGroup cbg)` - Creates a checkbox with a specified label and specified state, belonging to a specified checkbox group

Methods of the Checkbox class:

Method	Description
`addItemListener()`	Registers an `ItemListener` object to receive item events from the checkbox
`getCheckboxGroup()`	Returns the checkbox's group
`getLabel()`	Returns the checkbox's label
`getState()`	Determines if the checkbox is checked or unchecked
`setLabel()`	Sets the label of the check box with the specified string
`setState()`	Sets the specified checkbox state

Creating Radio Buttons (Mutually exclusive checkboxes):

- First create a CheckboxGroup instance – `CheckboxGroup cbg = new CheckboxGroup();`
- While creating the checkboxes, pass the checkbox group object as an argument to the constructor - `Checkbox (String s, boolean state, CheckboxGroup cbg)`

Choice:

Constructors

`Choice()` - Creates a new choice menu, and presents a pop-up menu of choices.

Methods of the Choice class:

Method	Description
`add()`	Adds an item to a choice menu
`addItem()`	Adds an item to a choice menu
`addItemListener()`	Registers an `ItemListener` object to receive item events from the Choice object
`getItem()`	Returns the item at the specified index as a string
`getItemCount()`	Returns the number of items in the choice menu
`getSelectedIndex()`	Returns the index number of the currently selected item
`getSelectedItem()`	Returns the currently selected item as a string
`insert()`	Inserts a specified item at a specified index position
`remove()`	Removes an item from the choice menu at the specified index

TextField:

Constructors

- `TextField()` - Creates a new text field
- `TextField(int cols)` - Creates a text field with the specified number of columns
- `TextField(String s)` – Creates a text field initialized with a specified string
- `TextField(String s, int cols)` - Creates a text field initialized with a specified string that is wide enough to hold a specified number of columns

Methods of the `TextField` class:

Method	Description
`isEditable()`	Returns a boolean value indicating whether or not a text field is editable
`setEditable()`	Passing True enables text to be edited, while False disables editing. The default is True.
`addActionListener()`	Registers an `ActionListener` object to receive action events from a text field
`getEchoChar()`	Returns the character used for echoing
`getColumns()`	Returns the number of columns in a text field

`setEchoChar()`	Sets the echo character for a text field
`getText()`	Returns the text contained in the text field
`setText()`	Sets the text for a text field

TextArea:

Constructors

· `TextArea()` - Creates a new text area
· `TextArea(int rows, int cols)` - Creates a new empty text area with specified rows and columns
· `TextArea(String s)` – Creates a new text area with the specified string
· `TextArea(String s, int rows, int cols)` - Creates a new text area with the specified string and specified rows and columns.
· `TextArea(String s, int rows, int cols, int scrollbars)` - Creates a text area with the specified text, and rows, columns, and scrollbar visibility as specified.

Methods of the TextArea class:

Method	**Description**
`getText()`	Returns the text contained in the text area as a string
`setText()`	Sets the specified text in the text area
`getRows()`	Returns the number of rows in the text area
`getColumns()`	Returns the number of columns in the text area
`selectAll()`	Selects all the text in the text area
`setEditable()`	A True value passed as an argument enables editing of the text area, while False disables editing. It is True by default.

List:

Constructors

· `List()` - Creates a new scrolling list
· `List(int rows)` - Creates a new scrolling list with a specified number of visible lines
· `List(int rows, boolean multiple)` - Creates a scrolling list to display a specified number of rows. A True value for Multiple allows multiple selection, while a False value allows only one item to be selected.

Methods of the List class:

Method	**Description**
`add()`	Adds an item to the end of the scrolling list
`addItemListener()`	Registers an `ItemListener` object to receive Item events from a scrolling list
`deselect()`	Deselects the item at the specified index position
`getItem()`	Returns the item at the specified index position
`getItemCount()`	Returns the number of items in the list
`getSelectedIndex()`	Returns the index position of the selected item
`getSelectedItem()`	Returns the selected item on the scrolling list
`isMultipleMode()`	Determines if the scrolling list allows multiple selection
`remove()`	Removes a list item from a specified position
`setMultipleMode()`	Sets a flag to enable or disable multiple selection

Scrollbar:

Constructors

▪ `Scrollbar()` - Creates a new vertical scroll bar
▪ `Scrollbar(int orientation)` - Creates a new scroll bar with a particular orientation, which is specified as `Scrollbar.HORIZONTAL` or `Scrollbar.VERTICAL`
▪ `Scrollbar(int orientation, int value, int visible, int minimum, int maximum)`- Creates a new scroll bar with the specified orientation, initial value, thumb size, minimum and maximum values

Methods of the Scrollbar class:

Method	**Description**
`addAdjustmentListener()`	Registers an adjustmentListener object to receive adjustment events from a scroll bar
`getBlockIncrement()`	Returns the block increment of a scrollbar as an integer.
`getMaximum()`	Returns the maximum value of a scrollbar as an integer
`getMinimum()`	Returns the minimum value of a scrollbar as an integer
`getOrientation()`	Returns the orientation of a scrollbar as an integer
`getValue()`	Returns the current value of a scrollbar as an integer

`setOrientation()`	Sets the orientation of a scrollbar
`setValue()`	Sets the current value of a scrollbar
`setMinimum()`	Sets the minimum value of a scrollbar
`setMaximum()`	Sets the maximum value of a scrollbar

Frame:

Constructors

- `Frame()` - Creates a new frame without any title
- `Frame(String s)` - Creates a new frame with the specified title

Menus:

- Can be added only to a frame
- A `MenuBar` instance is first created as:
  `MenuBar mb = new MenuBar();`
- The `MenuBar` instance is added to a frame using the `setMenuBar()` method of the `Frame` class as follows:
  `setMenuBar(mb);`
- Individual menus are created (instances of the `Menu` class) and added to the menu bar with the `add()` method

Dialog: Direct subclass of `java.awt.Window,` which accepts user input.

Constructors

- `Dialog(Frame parent, boolean modal)` – Creates a new initially invisible Dialog attached to the frame object parent. The second argument specifies whether the dialog box is Modal or Non-modal.
- `Dialog (Frame parent, String s, boolean modal)` – Same as the above. The second argument specifies the title of the dialog box.

FileDialog: Direct subclass of Dialog, which displays a dialog window for file selection.

Constructors

- `FileDialog(Frame f, String s)` - Creates a new dialog for loading files(file open dialog) attached to the frame with the specified title
- `FileDialog(Frame f, String s, int i)` - Creates a file dialog box with the specified title. The third argument specifies whether the dialog is for loading a file or saving a file. The value of i can be either `FileDialog.LOAD` or `FileDialog.SAVE`

AWT Event Listener interfaces: For every AWT event class there is a corresponding event-listener interface, which is a part of the `java.awt.event` package and provides the event-handling methods.

ActionListener interface: Implemented by a class that handles an action event. The method `actionPerformed()` must be overridden by the implementing class.

Interface method	Description
`actionPerformed()`	Invoked whenever an ActionEvent object is generated (button is clicked)

TextListener interface: Implemented by a class to handle text events. Whenever the text value of a component changes, an interface method called `textValueChanged` is invoked, which must be overridden in the implementing class.

Interface method	Description
`textValueChanged()`	Invoked whenever a `Text Event` object is generated (text value changes)

AdjustmentListener interface: Implemented by a class that handles adjustment events. The method `adjustmentValueChanged()`, overridden by the implementing class is invoked everytime an `AdjustmentEvent` object occurs (when a scrollbar is adjusted).

Interface method	Description
`adjustmentValueChanged()`	Invoked whenever an `AdjustmentEvent` object is generated (when a scrollbar thumb is adjusted)

ItemListener interface: Implemented to handle state change events. The method `itemStateChanged()`must be overridden by the implementing class.

Method	Description
`itemStateChanged()`	Invoked whenever an `ItemEvent` object is generated (a checkbox is checked, an item is selected from a choice menu, or an item is selected from a list)

FocusListener interface: Implemented to receive notifications whenever a component gains or loses focus. The two methods to be overridden are `focusGained()` and `focusLost()`. The corresponding adapter class is `FocusAdapter`.

Method	Description
`focusGained()`	Invoked whenever a component gains keyboard focus
`focusLost()`	Invoked whenever a component loses keyboard focus

KeyListener interface: Implemented to handle key events. Each of the three methods – `keyPressed()`, `keyReleased()`, `keyTyped()` – receives a `KeyEvent` object when a key event is generated.

Method	Description
`KeyPressed()`	Invoked whenever a key is pressed
`keyReleased()`	Invoked whenever a key is released

`keyTyped()`	Invoked whenever a key is typed

MouseListener interface: Implemented by a class handling mouse events. It comprises of five methods invoked when a `MouseEvent` object is generated. Its corresponding adapter class is the `MouseAdapter` class.

Method	Description
`mouseClicked()`	Invoked when mouse is clicked on a component
`mouseEntered()`	Invoked when mouse enters a component
`mouseExited()`	Invoked when mouse exits a component
`mousePressed()`	Invoked when mouse button is pressed on a component
`mouseReleased()`	Invoked when mouse button is released on a component

MouseMotionListener interface: Implemented by a class for receiving mouse-motion events. Consists of two methods – `mouseDragged()` and `mouseMoved()`, which is invoked when a `MouseEvent` object is generated. `MouseMotionAdapter` is its corresponding adapter class.

Method	Description
`mouseDragged()`	Invoked when the mouse is pressed on a component and dragged
`mouseMoved()`	Invoked when mouse is moved over a component

WindowListener interface: Implemented by a class to receive window events. It consists of seven different methods to handle the different kinds of window events, which are invoked when a `WindowEvent` object is generated. Its corresponding adapter class is the `WindowAdapter` class.

Method	Description
`windowOpened()`	Invoked when the window is made visible for the first time
`windowClosing()`	Invoked when the user attempts to close the window from the Windows system menu
`windowClosed()`	Invoked when the window has been closed as a result of calling the `dispose()` method
`windowActivated()`	Invoked when the window is made active i.e. the window can receive keyboard events

`windowDeactivated()`	Invoked when the window is no longer the active window i.e. the window can no longer receive keyboard events
`windowIconified()`	Invoked when a normal window is minimized
`windowDeiconified()`	Invoked when a minimized window is changed to normal state

java.sql.Driver interface: Implemented by every driver class.

Methods of the Driver interface:

Method	Description
`acceptsURL()`	Returns a Boolean value indicating whether the driver can open a connection to the specified URL
`connect()`	Tries to make a database connection to the specified URL
`getMajorVersion()`	Returns the driver's major version number
`getMinorVersion()`	Returns the driver's minor version number

jdbcCompliant()	Tests whether the driver is a genuine JDBC compliant driver

java.sql.Connection interface: Represents a session with a specific database. SQL statements are executed within a session and the results are returned.

Methods of the Connection interface:

Method	Description
`Close()`	Immediately releases the database and JDBC resources
`commit()`	Makes all changes since the last commit/rollback permanent, and releases the database locks held by the connection
`createStatement()`	Creates and returns a Statement object. It is used for sending SQL statements to be executed on the database
`getMetaData()`	Returns a `DatabaseMetaData` object that represents metadata about the database
`isReadOnly()`	Checks whether the connection is a read-only connection
`prepareCall()`	Creates and returns a `Callable Statement` object,

`prepareCall()`	Creates and returns a `CallableStatement` object (used for calling database stored procedures)
`prepareStatement()`	Creates and returns a `PreparedStatement` object (used for sending precompiled SQL statements to the database)
`rollback()`	Discards all the changes made since the last `commit/rollback` and releases database locks held by the connection
`setAutoCommit()`	Enables or disables the auto commit feature. It is disabled by default

java.sql.DriverManager class: Responsible for managing a set of JDBC drivers. It attempts to locate and load the JDBC driver specified by the `getConnection()` method.

Methods of the `DriverManager` class:

Method	**Description**
`getConnection()`	Attempts to establish a database connection with the specified database URL, and returns a Connection object
`getLoginTimeout()`	Returns the maximum number of seconds a driver can wait when attempting to connect to the database

`registerDriver()`	Registers the specified driver with the `DriverManager`
`setLoginTimeout()`	Sets the maximum number of seconds a driver can wait when attempting to connect to the database
`getDrivers()`	Returns an enumeration of all the drivers installed on the system
`getDriver()`	Returns a Driver object that supports connection through a specified URL