## UNIT-2: CLASSES, INHERITANCE, EXCEPTIONS, APPLETS

### 1. Defining Classes, Class Name

- To define a class, use the class keyword and the name of the class:

  class MyClassName {

  **... }**

- If this class is a subclass of another class, use extends to indicate the superclass of this

  class:

  class myClassName extends mySuperClassName {

  ...

  }

- If this class implements a specific interface, use implements to refer to that interface: class MyRunnableClassName implements Runnable {

...

}

**Super Classes**

- Each class has a superclass (the class above it in the hierarchy), and each class can have one or more subclasses (classes below that class in the hierarchy). Classes further down in the hierarchy are said to inherit from classes further up in the hierarchy.
- Subclasses inherit all the methods and variables from their superclasses—that is, in any particular class, if the superclass defines behavior that your class needs, you don't have to redefine it or copy that code from some other class. Your class

automatically gets that behavior from its superclass, that superclass gets behavior from its superclass, and so on all the way up the hierarchy.

- **At the top of the Java class hierarchy is the class Object;** all classes inherit from this one superclass. Object is the most general class in the hierarchy; it defines behavior inherited by all the classes in the Java class hierarchy. Each class farther down in the hierarchy adds more information and becomes more tailored to a specific purpose.

**E**.g.

public class HelloAgainApplet extends java.applet.Applet { }

## 2. Constructors, Creating instances of a class

The following example demonstrates the creation of classes, creating objects and the usage of constructors

class Motorcycle { }

create some instance variables for this class

String make; String color; boolean engineState;

Add some behavior (methods) to the class.

void startEngine() {

if (engineState == true)

System.out.println("The engine is already on."); else {

engine State = true;

System.out.println("The engine is now on."); }

}

The program looks like this now :

```
class Motorcycle {
String make;
String color;
boolean engineState;
void startEngine() {
if (engineS tate == true)
System.out.println("The engine is already on.");
engine is now on.");
}
```

The showAtts method prints the current values of the instance variables in an instance of your Motorcycle class. Here's what it looks like:

void showAtts() {

System. out .println ("This motorcycle is a " + color + " " + make);

if (engineState == true)

System.out.println("The engine is on."); else System.out.println("The engine is off.");

The showAtts method prints two lines to the screen: the make and color of the motorcycle object, and whether or not the engine is on or off.

**2.1 Add the main method**

public static void main (String args[]) { Motorcycle m = new Motorcycle(); m.make = "Yamaha RZ350";

m.color = "yellow";

System.out.println("Calling showAtts..."); m.showAtts(); System.out.println("———

—"); System.out.println("Starting

engine..."); m.startEngine();

System.out.println("————");

System.out.println("Calling showAtts..."); m.showAtts(); System.out.println("———

—"); System.out.println("Starting

engine..."); m.startEngine();

}

With the main() method, the Motorcycle class is now an application, and you can compile it again and this time it'll run. Here's how the output should look:

Calling showAtts...

This motorcycle is a yellow Yamaha RZ350 The engine is off.

Starting engine... The engine is now on.

Calling showAtts...

This motorcycle is a yellow Yamaha RZ350 The engine is on.

Starting engine...

The engine is already on.

## 3. Inheritance

Inheritance is a powerful mechanism that means when you write a class you only have to specify how that class is different from some other class; inheritance will give you automatic access to the information contained in that other class.

With inheritance, all classes—those you write, those from other class libraries that you use, and those from the standard utility classes as well—are arranged in a strict hierarchy

### 3.1 Single and Multiple Inheritance

Single inheritance means that each Java class can have only one superclass (although any given superclass can have multiple subclasses).

In other object-oriented programming languages, such as C++, classes can have more than one superclass, and they inherit combined variables and methods from all those classes. This is called multiple inheritance.

Multiple inheritance can provide enormous power in terms of being able to create classes that factor just about all imaginable behavior, but it can also significantly complicate class definitions and the code to produce them. **Java makes inheritance simpler by being only singly inherited.**

### 3.2 Overriding Methods

- When a method is called on an object, Java looks for that method definition in the class of that object, and if it doesn't find one, it passes the method call up the class hierarchy until a method definition is found.
- Method inheritance enables you to define and use methods repeatedly in subclasses without having to duplicate the code itself.
- However, there may be times when you want an object to respond to the same methods but have different behavior when that method is called. In this case, you can override that method. Overriding a method involves defining a method in a subclass that has the same signature as a method in a superclass. Then, when that method is called, the method in the subclass is found and executed instead of the one in the superclass.

**3.3 Creating Methods that Override Existing Methods**

To override a method, all you have to do is create a method in your subclass that has the same signature (name, return type, and parameter list) as a method defined by one of your class's superclasses. Because Java executes the first method definition it finds that matches the signature, this effectively "hides" the original method definition. Here's a simple example

**The PrintClass class.**

```
class PrintClass {

int x = 0; int y = 1;

void printMe() {

System.out.println("X is " + x + ", Y is " + y);

System.out.println("I am an instance of the class " +

this.getClass().getName());
```

Create a class called PrintSubClass that is a subclass of (extends) PrintClass.

```
class PrintSubClass extends PrintClass { int z = 3;

public static void main(String args[]) { PrintSubClass obj = new PrintSubClass(); obj
.printMe();

}

}
```

Here's the output from PrintSubClass:

X is 0, Y is 1

I am an instance of the class PrintSubClass

In the main() method of PrintSubClass, you create a PrintSubClass object and call the printMe() method. Note that PrintSubClass doesn't define this method, so Java looks for it in each of PrintSubClass's superclasses—and finds it, in this case, in PrintClass. because printMe() is still defined in PrintClass, it doesn't print the z instance variable.

To call the original method from inside a method definition, use the super keyword to pass the method call up the hierarchy:

void myMethod (String a, String b) { // do stuff here

super.myMethod(a, b);

// maybe do more stuff here }

The super keyword, somewhat like the this keyword, is a placeholder for this class's superclass. You can use it anywhere you can use this, but to refer to the superclass rather than to the current class.

## 4. Exception handling

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

### 4.1 The Three Kinds of Exceptions

- Checked exceptions *are subject* to the Catch or Specify Requirement. All exceptions are checked exceptions, except for those indicated by Error, RuntimeException, and their subclasses.

- Errors *are not subject* to the Catch or Specify Requirement. Errors are those exceptions indicated by Error and its subclasses.

- Runtime exceptions *are not subject* to the Catch or Specify Requirement. Runtime exceptions are those indicated by Runtime Except ion and its subclasses.

Valid Java programming language code must honor the *Catch or Specify Requirement.* This means that code that might throw certain exceptions must be enclosed by either of the following:

• A try statement that catches the exception. The try must provide a handler for the exception, as described in Catching and Handling Exceptions.

• A method that specifies that it can throw the exception. The method must provide a throws clause that lists the exception, as described in Specifying the Exceptions Thrown by a Method.

Code that fails to honor the Catch or Specify Requirement will not compile.

This example describes how to use the three exception handler components — the try, cat ch, and finally blocks

**4.2 try block**

• The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block. In general, a try block looks like the following.

```
try {
code
}
catch and finally blocks . . .
```

Example :

```
private Vector vector;
private static final int SIZE = 10;
PrintWriter out = null;
try {
System.out.println("Entered try statement");
out = new PrintWriter(new FileWriter("OutFile.txt"));
for (int i = 0; i < SIZE; i++) {
out.println("Value at: " + i + " = " + vector.elementAt(i));
}
}
```

**The catch Blocks**

You associate exception handlers with a try block by providing one or more catch blocks directly after the try block. No code can be between the end of the try block and the beginning of the first catch block.

try {

} catch *(ExceptionType name) {*

*}* catch *(ExceptionType name) {*

*}*

Each catch block is an exception handler and handles the type of exception indicated by its argument

**finally block**

The runtime system always executes the statements within the finally block regardless of what happens within the try block. So it's the perfect place to perform cleanup.

The following finally block for the write Li st method cleans up and then closes the PrintWriter.

finally {

if (out != null) {

System.out.println("Closing PrintWriter"); out . close();

} else {

System.out.println("PrintWriter not open"); }

}

## 5. The Applet Class

**Applet Basics**

- An applet is a special kind of Java program that a browser enabled with Java technology can download from the internet and run.
- An applet is typically embedded inside a web-page and runs in the context of the browser.
- An applet must be a subclass of the java.applet.Applet class, which provides the standard interface between the applet and the browser environment.
- Simple example :

public class HelloWorld extends java.applet.Applet {

public void paint(java.awt.Graphics g) {

g.drawString("Hello World!",50,25); System.out.println("Hello World!");

}

}

An applet can be included in an HTML page, much in the same way an image is included in a page.

When Java technology enabled Browser is used to view a page that contains an applet, the applet's code is transferred to your system and executed by the browser's Java Virtual Machine (JVM)

**Two Types of Applets**

1 .Local applet - operate in single machine browser which is not connected in network, 2.Remote applet - remote applet operate over internet via network.

**Applet Architecture**

Event driven :

An applet waits until an event occurs.

The *AWT* notifies the applet about an event by calling event handler that has been provided by the applet.

The applet takes appropriate action and then quickly return control to *AWT* All *Swing* components descend from the AWT Container class

User initiates interaction with an Applet *(and not the other way around)* **An Applet Skeleton**

import java.awt.*;

import javax.swing.*;

/*

<applet code="AppletSkel" width=300 height=100>

</applet>

*/

public class AppletSkel extends JApplet { // Called first.

public void init() {

```
// initialization
}
/* Called second, after init(). Also called whenever the applet is restarted. */
public void start() {
// start or resume execution
}
// Called when the applet is stopped. public void stop () {
// suspends execution
}
/* Called when applet is terminated. This is the last
method executed. */
public void destroy() {
// perform shutdown activities }
// Called when an applet's window must be restored. public void paint(Graphics g) {
// redisplay contents of window
}
}
```

**5.1 Simple Applet Display Methods**

void drawstring(String message, int x, int y) - void setBackground(Color newColor) void setForeground(Color newColor) **Example :**

```
public class SimpleApplet extends Applet { public void paint (Graphics g) { g.drawString("First Applet", 50, 50 ); }
}
```

**Requesting Repainting**

repaint( ) function is called when you have changed something and want your changes to show up on the screen

repaint( ) is a *request*--it might not happen

When you call repaint( ), Java schedules a call to update(Graphics g)

Here's what update does:

public void update(Graphics g) {

// Fills applet with background color, then

paint(g);

**Using The Status Window**

Syntax : public void **showStatus**(String status)

**Parameters:**

status - a string to display in the status window.

Requests that the argument string be displayed in the "status window".

Many browsers and applet viewers provide such a window, where the application can inform users of its current state.

**Example :**

import java.applet. *; import java.awt.*;

public class NetExample extends Applet

{

private AppletContext browser = null;

private Button showStatus = new Button("Show Status"); public void init()

{

Panel panel = new Panel();

panel.setLayout(new GridLayout(1 ,2));

panel.add(showStatus);

setLayout(new BorderLayout()); add("South", panel);

browser = getAppletContext(); }

public boolean action(Event e, Object o)

{

```
if (e.target == showStatus)

browser.showStatus("Here is something for your status line ..."); return true;

}

}
```

## 6. The HTML Applet Tag

- The APPLET tag is used to start an applet from both an HTML document and from an applet viewer.
- An applet viewer will execute each APPLET tag that it finds in a separate window,while web browsers like Netscape Navigator, Internet Explorer, and HotJava will allow many applets on a single page.
- The syntax for the standard APPLET tag is shown here. Bracketed items are optional.

< APPLET

[CODEBASE = *codebaseURL]*

CODE = *appletFile*

[ALT = *alte rnate Text]*

[NAME = *appletInstanceName]*

WIDTH = *pixels* HEIGHT = *pixels*

[ALIGN = *alignment]*

[VSPACE = *pixels]* [HSPACE = *pixels]*

>

[< PARAM NAME = *AttributeName* VALUE = *Attribute Value*>] [< PARAM NAME = *AttributeName2* VALUE = *Attribute Value*>] . . .

*[HTML Displayed in the absence of Java]*

</APPLET>

- **CODEBASE** is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag). The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read.

- **CODE** is a required attribute that gives the name of the file containing your applet's compiled **.class** file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.
- **ALT** is an optional attribute used to specify a short text message that should be displayed if the browser understands the APPLET tag but can't currently run Java applets. This is distinct from the alternate HTML you provide for browsers that don't support applets.
- **WIDTH AND HEIGHT** are required attributes that give the size (in pixels) of the applet display area.
- **ALIGN** is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values:
- LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

**VSPACE AND HSPACE** These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes.

**PARAM NAME AND VALUE** The PARAM tag allows you to specify appletspecific arguments in an HTML page. Applets access their attributes with the **getParameter( )** method.

**Passing Parameters to Applets**

Parameters are passed to applets in NAME=VALUE pairs in <PARAM> tags between the opening and closing APPLET tags.

• Inside the applet, you read the values passed through the PARAM tags with the getParameter() method of the java.applet.Applet class.

The applet parameter "Message" is the string to be drawn.

import java.applet. *; import
java.awt.*;

```
public class DrawStringApplet extends Applet {
 private String defaultMessage = "Hello!"; public
 void paint(Graphics g) {

String inputFromPage = this .getParameter("Mes
 sage");

   if (inputFromPage == null) inputFromPage = defaultMessage;
   g.drawString(inputFromPage, 50, 25);

  }
}
```

HTML file that references the above applet.

<HTML> <HEAD>

<TITLE> Draw String </TITLE>

</HEAD>

<BODY>

This is the applet:<P>

<APPLET code="DrawStringApplet" width="3 00" height="50">

<PARAM name="Message" value="Howdy, there!"> This page will be very boring if your

browser doesn't understand Java.

</APPLET>

</BODY> </HTML>

**getDocumentBase() and getCodeBase()**

Syntax : public URL **getDocumentBase()**

**Returns:**

the URL of the document that contains this applet.

- Gets the URL of the document in which this applet is embedded.
- For example, suppose an applet is contained within the document:

http://java.sun.com/products/jdk/1.2/index.html

- The document base is:

  http://java.sun.com/products/jdk/1.2/index.html

  Syntax : public URL **getCodeBase()**

  **Returns:**

  the base URL of the directory which contains this applet.

- Gets the base URL. This is the URL of the directory which contains this applet.
- Example segments:

URL codeBase = getCodeBase();

Image myImage = getImage(codeBase, "images/myimage.gif"); **Applet Context and showDocument()**

AppletContext is an interface that provides the means to control the browser environment in which the applet is running.

**The AudioClip Interface**

- The AudioC lip interface is a simple abstraction for playing a sound clip.
- Multiple Audi oC lip items can be playing at the same time, and the resulting sound is mixed together to produce a composite.
- It has the following methods :

**play**

public abstract void play()

- **loop** Starts playing this audio clip. Each time this method is called, the clip is restarted from the beginning.
- **stop**

public abstract void loop()

public abstract void stop()

Stops playing this audio clip.

**The AppletStub Interface**

The AppletStub interface provides a way to get information from the run-time browser environment.

The Applet class provides methods with similar names that call these methods. Methods

*public abstract boolean isActive ()*

The isActive() method returns the current state of the applet. While an applet is initializing, it is not active, and calls to isActive() return false. The system marks the applet active just prior to calling start(); after this point, calls to isActive() return true.

- *public abstract URL getDocumentBase ()*

The getDocumentBase() method returns the complete URL of the HTML file that loaded the applet. This method can be used with the getImage() or getAudioClip() methods to load an image or audio file relative to the HTML file.

- *public abstract URL getCodeBase ()*

The getCodeBase() method returns the complete URL of the *.class* file that contains the applet. This method can be used with the getImage() method or the getAudioClip() method to load an image or audio file relative to the *.class* file.

- *public abstract String getParameter (String name)*

The getParameter() method allows you to get parameters from <PARAM> tags within the <APPLET> tag of the HTML file that loaded the applet. The name parameter of getParameter() must match the name string of the <PARAM> tag; name is case insensitive. The return value of getParameter() is the value associated with name; it is always a String regardless of the type of data in the tag. If name is not found within the <PARAM> tags of the <APPLET>, getParameter() returns null.

- *public abstract AppletContext getAppletContext ()*

The getAppletContext() method returns the current AppletContext of the applet. This is part of the stub that is set by the system when setStub() is called.

- *public abstract void appletResize (int width, int height)*

The appletResize() method is called by the resize method of the Applet class. The method changes the size of the applet space to width x height. The browser must support changing the applet space; if it doesn't, the size remains unchanged

**Output To the Console**

The drawString method can be used to output strings to the console. The position of the text can also be specified.

The following prog shows this concept:

```
public class ConsolePrintApplet1 extends java.applet.Applet
{
```

```
public void init () {
// Put code between this line
double x = 5.0; double y = 3.0;
System.out.println( "x * y = "+ (x*y) );
System.out.println( "x / y = "+ (x/y) );
                    //          // and this line.
}
// Paint message in the applet window. Public
```

UNIT-3 : **MULTI THREADED PROGRAMMING, EVENT HANDLING**

## 1. What are Threads?

A thread is a single path of execution of code in a program.

- A Multithreaded program contains two or more parts that can run concurrently.
- Each part of such a program is called a Thread.
- Each thread defines a separate path of execution. Multithreading is a specialized form of Multitasking.

### 1.1 How to make the classes threadable

A class can be made threadable in one of the following ways