BCA PART- III **PAPER : BCA-302**

JAVA PROGRAMMING

UNIT NO. 2

Lesson Nos :

Department of Distance Education
Punjabi University, Patiala

Lesson No. 2.1 **Author : Kanwal Preet Singh**

Introduction to Classes

2.1.1 Introduction

In classic, procedural programming you try to make the real world problem you're attempting to solve fit a few, predetermined data types: integers, floats, Strings and arrays perhaps. In object oriented programming you create a model for a real world system. Classes are programmer-defined types that model the parts of the system. A *class* is a programmer defined type that serves as a blueprint for instances of the class. You can still have ints, floats, Strings, and arrays; but you can also have cars, motorcycles, people, buildings, clouds, dogs, angles, students, courses, bank accounts and any other type that's important to your problem.

Classes specify the data and behavior possessed both by themselves and by the objects built from them. A class has two parts: the fields and the methods. Fields describe what the class is. Methods describe what the class does. Using the blueprint provided by a class, you can create any number of objects, each of which is called an ***instance* of** the class. Different objects of the same class have the same fields and methods, but the values of the fields will in general differ. For example, all people have eye color; but the color of each person's eyes can be different from others. On the other hand, objects have the same methods as all other objects in the class except in so far as the methods depend on the value of the fields and arguments to the method. This dichotomy is

reflected in the runtime form of objects. Every object has a separate block of memory to store its fields, but the bytes in the methods are shared between all objects in a class.

Following the principles of Object Oriented Programming (OOP), everything in Java is either a class, a part of a class or describes how a class behaves. Objects are the physical instantiations of classes. They are living entities within a program that have independent lifecycles and that are created according to the class that describes them. Just as many buildings can be built from one blueprint, many objects can be instantiated from one class. Many objects of different classes can be created, used, and destroyed in the course of executing a program. Programming languages provide a number of simple data types like int, float and String. However very often the data you want to work with may not be simple ints, floats or Strings. Classes let programmers define their own more complicated data types.

All the action in Java programs takes place inside class blocks. In Java almost everything of interest is either a class itself or belongs to a class. Methods are defined inside the classes they belong to. Even basic data primitives like integers often need to be incorporated into classes before you can do many useful things with them. The class is the fundamental unit of Java programs.

2.1.2 Objective

After reading this lesson you will be able to understand:

- Classes
- Objects
- Object References
- Member Variables
- Methods
- Passing arguments to methods

2.1.3 Introduction to Java Classes

A class is nothing but a blueprint or a template for creating different objects which defines its properties and behaviors. Java class objects exhibit the properties and behaviors defined by its class. A class can contain fields and methods to describe the behavior of an object.

Methods are nothing but members of a class that provide a service for an object or perform some business logic. Java fields and member functions names are case sensitive. Current states of a class's corresponding object are stored in the object's instance variables. Methods define the operations that can be performed in java programming. A class has the following general syntax:

//Contents of SomeClassName.java

[public] [(abstract | final)] class SomeClassName [extends SomeParentClass] [implements SomeInterfaces]

```
{
    // variables and methods are declared within the curly braces
}
```

- A class can have public or default (no modifier) visibility. The ***visibility*** indicates scope or accessibility from other objects. ***public*** means visible everywhere. The default (ie. omitted) is ***package*** friendly or visible within the current package only.
- The second optional group indicates the capability of a class to be **inherited** or extended by other classes. It can be either abstract, final or concrete (no modifier). ***abstract*** classes must be extended and ***final*** classes can never be extended by inheritance. The default (ie. omitted) indicates that the class may or may not be extended at the programmers discretion.
- It must have the class keyword and class must be followed by a legal identifier.
- It may optionally extend one parent class. By default, it will extend java.lang.Object.
- It may optionally implement any number of comma-separated interfaces.
- The class's variables and methods are declared within a set of curly braces '{}'.
- Each .java source file may contain only one public class. A source file may contain any number of default visible classes.
- Finally, the source file name must match the public class name and it must have a .java suffix.

Here is an example of a Horse class. Horse is a subclass of Mammal, and it implements the Hoofed interface.

```
public class Horse extends Mammal implements Hoofed
{
    //Horse's variables and methods go here
}
```

Example : The Car Class

Suppose you need to write a traffic simulation program that watches cars going past an intersection. Each car has a speed, a maximum speed and a license plate that uniquely identifies it. In traditional programming languages you'd have two floating point and one string variable for each car. With a class you combine these into one thing like this.

```
class Car
 {
  String licensePlate; // e.g. "PB11R2300"
  double speed;        // in kilometers per hour
  double maxSpeed;     // in kilometers per hour
 }
```

These variables (licensePlate, speed and maxSpeed) are called the *member variables*, i*nstance variables* or *fields* of the class. Fields tell you what a class is and what its properties are.

An ***object*** is a specific instance of a class with particular values for the fields. While a class is a general blueprint for objects, an instance is a particular object.

2.1.4 Constructing objects with new

To instantiate an object in Java, use the keyword new followed by a call to the class's constructor. First declare an instance variable of the class as follows:

Class_name instance_variable;

Then create an object of the class using new operator and assign it to the instance variable declared above as:

instance_variable = new class_name();

For example Here's how you'd create a new Car variable called c and the assign to it an object of Class **Car**:

```
Car c;
c = new Car();
```

The first word, Car, declares the type of the variable c. Classes are types and variables of a class type need to be declared just like variables that are ints or doubles. The equals sign is the assignment operator and new is the construction operator. Finally notice the Car() method. The parentheses tell you this is a method and not a data type like the Car on the left hand side of the assignment. This is a constructor, a method that creates a new instance of a class. You'll learn more about constructors shortly. However if you do nothing, then the compiler inserts a default constructor that takes no arguments. This is often condensed into one line like this:

```
Car c = new Car();
```

2.1.5 Object References

In Java, a class is a type, similar to the built-in types such as int and boolean. So, a class name can be used to specify the type of a variable in a declaration statement, the type of a formal parameter or the return type of a function. For example, a program could define a variable named std of type Student with the statement

```
Student std;
```

However, declaring a variable does not create an object! This is an important point, which is related to this Very Important Fact:

In Java, no variable can ever hold an object. A variable can only hold a reference to an object.

You should think of objects as floating around independently in the computer's memory. In fact, there is a special portion of memory called the heap where objects live. Instead of holding an object itself, a variable holds the information necessary to find the object in memory. This information is called a reference or pointer to the

object. In effect, a reference to an object is the address of the memory location where the object is stored. When you use a variable of class type, the computer uses the reference in the variable to find the actual object. In a program, objects are created using an operator called new, which creates an object and returns a reference to that object. For example, assuming that std is a variable of type Student, declared as above, the assignment statement:

std = new Student();

would create a new object which is an instance of the class Student, and it would store a reference to that object in the variable std. The value of the variable is a reference to the object, not the object itself. It is not quite true, then, to say that the object is the "value of the variable std" (though sometimes it is hard to avoid using this terminology). It is certainly not at all true to say that the object is "stored in the variable std." The proper terminology is that "the variable std refers to the object," and we will try to stick to that terminology as much as possible. So, suppose that the variable std refers to an object belonging to the class Student. That object has instance variables name, test1, test2 and test3. These instance variables can be referred to as std.name, std.test1, std.test2 and std.test3. This follows the usual naming convention that when B is part of A, then the full name of B is A.B. For example, a program might include the lines

```
System.out.println("Hello, " + std.name + ". Your test grades are:");
System.out.println(std.test1);
System.out.println(std.test2);
System.out.println(std.test3);
```

This would output the name and test grades from the object to which std refers. Similarly, std can be used to call the getAverage() instance method in the object by saying std.getAverage(). To print out the student's average, you could say:

```
System.out.println( "Your average is " + std.getAverage() );
```

More generally, you could use std.name any place where a variable of type String is legal. You can use it in expressions. You can assign a value to it. You can even use it to call subroutines from the String class. For example, std.name.length() is the number of characters in the student's name. It is possible for a variable like std, whose type is given by a class, to refer to no object at all. We say in this case that std holds a null reference. The null reference is written in Java as "null". You can store a null reference in the variable std by saying

std = null;

and you could test whether the value of std is null by testing

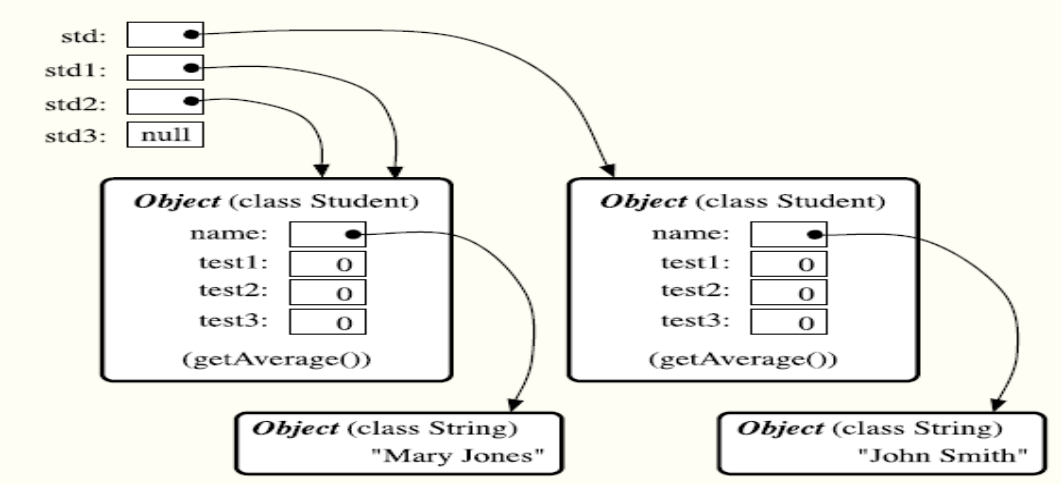
if (std == null) . . .

If the value of a variable is null, then it is, of course, illegal to refer to instance variables or instance methods through that variable—since there is no object, and

hence no instance variables to refer to. For example, if the value of the variable std is null, then it would be illegal to refer to std.test1. If your program attempts to use a null reference illegally like this, the result is an error called a null pointer exception. Let's look at a sequence of statements that work with objects:

```
Student std, std1, // Declare four variables of
std2, std3; // type Student.
std = new Student(); // Create a new object belonging
// to the class Student, and
// store a reference to that
// object in the variable std.
std1 = new Student(); // Create a second Student object
// and store a reference to
// it in the variable std1.
std2 = std1; // Copy the reference value in std1
// into the variable std2.
std3 = null; // Store a null reference in the
// variable std3.
std.name = "John Smith"; // Set values of some instance variables.
std1.name = "Mary Jones";
// (Other instance variables have default
// initial values of zero.)
```

After the computer executes these statements, the situation in the computer's memory looks like this:



This picture shows variables as little boxes, labeled with the names of the variables. Objects are shown as boxes with round corners. When a variable contains a reference to an object, the value of that variable is shown as an arrow pointing to the

object. The variable std3, with a value of null, doesn't point anywhere. The arrows from std1 and std2 both point to the same object. This illustrates a Very Important Point:

When one object variable is assigned to another, only a reference is copied. The object referred to is not copied.

When the assignment "std2 = std1;" was executed, no new object was created. Instead, std2 was set to refer to the very same object that std1 refers to. This has some consequences that might be surprising. For example, std1.name and std2.name are two different names for the same variable, namely the instance variable in the object that both std1 and std2 refer to. After the string "Mary Jones" is assigned to the variable std1.name, it is also be true that the value of std2.name is "Mary Jones". There is a potential for a lot of confusion here, but you can help protect yourself from it if you keep telling yourself, "The object is not in the variable. The variable just holds a pointer to the object." You can test objects for equality and inequality using the operators == and !=, but here again, the semantics are different from what you are used to. When you make a test "if (std1 == std2)", you are testing whether the values stored in std1 and std2 are the same. But the values are references to objects, not objects. So, you are testing whether std1 and std2 refer to the same object, that is, whether they point to the same location in memory. This is fine, if its what you want to do. But sometimes, what you want to check is whether the instance variables in the objects have the same values. To do that, you would need to ask whether

"std1.test1 == std2.test1 && std1.test2 == std2.test2 &&
std1.test3 == std2.test3 && std1.name.equals(std2.name)".

2.1.6 The Member Access Separator

Once you've constructed a car, you want to do something with it. To access the fields of the car you use the . separator. The Car class has three fields

- licensePlate
- speed
- maxSpeed

Therefore if c is a Car object, c has three fields as well:

- c.licensePlate
- c.speed
- c.maxSpeed

You use these just like you'd use any other variables of the same type. For instance:

```
Car c = new Car();
c.licensePlate = " PB11R2300";
c.speed = 64.0;
c.maxSpeed = 130.00;
System.out.println(c.licensePlate + " is moving at " + c.speed +
"kilometers per hour.");
```

The . separator selects a specific member of a Car object by name.

Using a Car object in a different class

The next program creates a new car, sets its fields, and prints the result:

```
class CarTest
 {
  public static void main(String args[])
   {
    Car c = new Car();
    c.licensePlate = " PB11R2300";
    c.speed = 64.0;
    c.maxSpeed = 130.00;
    System.out.println(c.licensePlate + " is moving at " + c.speed +
    " kilometers per hour.");
   }
 }
```

This program requires not just the CarTest class but also the Car class. To make them work together put the Car class in a file called Car.java. Put the CarTest class in a file called CarTest.java. Put both these files in the same directory. Then compile both files in the usual way. Finally run CarTest. For example,

```
% javac Car.java
% javac CarTest.java
% java CarTest
```

PB11R2300 is moving at 64.0 kilometers per hour.

Note that Car does not have a main() method so you cannot run it. It can exist only when called by other programs that do have main() methods. Many of the applications you write from now on will use multiple classes. It is customary in Java to put every class in its own file. Later you'll learn how to use packages to organize your commonly used classes in different directories. For now keep all your .java source code and .class byte code files in one directory.

Initializing Fields

Fields can (and often should) be initialized when they're declared, just like local variables.

```
class Car
 {
  String licensePlate = "";    // e.g. "PB11R2300"
  double speed        = 0.0;   // in kilometers per hour
  double maxSpeed     = 130.0; // in kilometers per hour
}
```

The next program creates a new car and prints it:

```
class CarTest2
 {
  public static void main(String[] args)
   {
     Car c = new Car();
     System.out.println(c.licensePlate + " is moving at " + c.speed +
     "kilometers per hour.");
   }
 }
```

For example,

$ javac Car.java

$ javac CarTest2.java

$ java CarTest

Output is:

is moving at 0.0 kilometers per hour.

2.1.7 Methods

Data types aren't much use unless you can do things with them. For this purpose classes have methods. Fields say what a class *is*. Methods say what a class *does*. The fields and methods of a class are collectively referred to as the *members of the class*.

The classes you've encountered up till now have mostly had a single method, main(). However, in general classes can have many different methods that do many different things. For instance the Car class might have a method to make the car go as fast as it can. For example,

```
class Car
 {
   String licensePlate = "";    // e.g. "PB11R2300"
   double speed       = 0.0;   // in kilometers per hour
   double maxSpeed    = 130.0; // in kilometers per hour
   // accelerate to maximum speed
   // put the pedal to the metal
   void floorIt()
    {
      this.speed = this.maxSpeed;
    }
 }
```

The fields are the same as before, but now there's also a method called floorIt(). It begins with the Java keyword void which is the return type of the method. Every method must have a return type which will either be void or some data type like int, byte, float or String. The return type says what kind of the value will be sent back to

the calling method when all calculations inside the method are finished. If the return type is int, for example, you can use the method anywhere you use an int constant. If the return type is void then no value will be returned.

floorIt is the name of this method. The name is followed by two empty parentheses. Any arguments passed to the method would be passed between the parentheses, but this method has no arguments. Finally an opening brace ({) begins the body of the method. There is one statement inside the method

this.speed = this.maxSpeed;

Notice that within the Car class the field names are prefixed with the keyword this to indicate that I'm referring to fields in the current object.

Finally the floorIt() method is closed with a } and the class is closed with another }.

Invoking Methods

Outside the Car class, you call the floorIt() method just like you reference fields, using the name of the object you want to accelerate to maximum and the . separator as demonstrated below

```
class CarTest3
 {
  public static void main(String args[])
   {
     Car c = new Car();
     c.licensePlate = " PB11R2300";
     c.maxSpeed = 130.0;
     System.out.println(c.licensePlate + " is moving at " + c.speed +
     " kilometers per hour.");
    c.floorIt();
    System.out.println(c.licensePlate + " is moving at " + c.speed +
     " kilometers per hour.");
   }
 }
```

The output is:

PB11R2300is moving at 0.0 kilometers per hour.

PB11R2300is moving at 130.0 kilometers per hour.

The floorIt() method is completely enclosed within the Car class. Every method in a Java program must belong to a class. Unlike C++ programs, Java programs cannot have a method hanging around in global space that does everything you forgot to do inside your classes.

Implied this

```
class Car
 {
```

```
 String licensePlate = "";     // e.g. "PB11R2300"
 double speed        = 0.0;    // in kilometers per hour
 double maxSpeed     = 130.0; // in kilometers per hour
 void floorIt()
  {
    speed = maxSpeed;
  }
}
```

Within the Car class, you don't absolutely need to prefix the field names with this. like this.licensePlate or this.speed. Just licensePlate and speed are sufficient. The this. may be implied. That's because the floorIt() method must be called by a specific instance of the Car class, and this instance knows what its data is. Or another way of looking at it, the every object has its own floorIt() method.

For clarity, we will use an explicit this, and I recommend you do so too, at least initially. As you become more comfortable with Java, classes, references, and OOP, you will be able to leave out the this without fear of confusion. Most real-world code does not use an explicit this.

2.1.8 Member Variables vs. Local Variables

```
class Car
 {
  String licensePlate = "";     // member variable
  double speed;       = 0.0;    // member variable
  double maxSpeed;    = 130.0; // member variable
  boolean isSpeeding()
   {
    double excess;    // local variable
    excess = this.maxSpeed - this.speed;
    if (excess < 0) return true;
    else return false;
   }
 }
```

Until now all the programs you've seen were quite simple in structure. Each had exactly one class. This class had a single method, main(), which contained all the program logic and variables. The variables in those classes were all local to the main() method. They could not be accessed by anything outside the main() method. These are called **local variables.**

This sort of program is the amoeba of Java. Everything the program needs to live is contained inside a single cell. It's quite an efficient arrangement for small

organisms, but it breaks down when you want to design something bigger or more complex.

The licensePlate, speed and maxSpeed variables of the Car class, however, belong to a Car object, not to any individual method. They are defined outside of any methods but inside the class and are used in different methods. They are called **member variables or fields**.

Member variable, instance variable, and field are different words that mean the same thing. *Field* is the preferred term in Java. Member variable is the preferred term in C++. A member is not the same as a member variable or field. Members include both fields and methods.

2.1.9 Passing Arguments to Methods

It's generally considered bad form to access fields directly. Instead it is considered good object oriented practice to access the fields only through methods. This allows you to change the implementation of a class without changing its interface. This also allows you to enforce constraints on the values of the fields.

To do this you need to be able to send information into the Car class. This is done by passing arguments. For example, to allow other objects to change the value of the speed field in a Car object, the Car class could provide an accelerate() method. This method does not allow the car to exceed its maximum speed, or to go slower than 0 kph.

```
void accelerate(double sp)
 {
   this.speed = this.speed + sp;
   if (this.speed > this.maxSpeed)
    {
      this.speed = this.maxSpeed;
    }
   if (this.speed <  0.0)
    {
      this.speed = 0.0;
    }
 }
```

The first line of the method is called its *signature*. The signature

void accelerate(double sp)

indicates that accelerate() returns no value and takes a single argument, a double which will be referred to as sp inside the method. sp is a purely *formal* argument. Java passes method arguments by value, not by reference.

Passing Arguments to Methods, An Example

class Car

```
{
 String licensePlate = "";     // e.g. "PB11R2300"
 double speed       = 0.0;    // in kilometers per hour
 double maxSpeed    = 130.0; // in kilometers per hour

 // accelerate to maximum speed
 // put the pedal to the metal
 void floorIt()
  {
    this.speed = this.maxSpeed;
  }
 void accelerate(double sp)
  {
    this.speed = this.speed + sp;
    if (this.speed > this.maxSpeed)
     {
       this.speed = this.maxSpeed;
     }
    if (this.speed <  0.0)
     {
       this.speed = 0.0;
     }
  }
}
class CarTest4
 {
   public static void main(String[] args)
    {
     Car c = new Car();
     c.licensePlate = " PB11R2300";
     c.maxSpeed = 130.0;
     System.out.println(c.licensePlate + " is moving at " + c.speed +
     " kilometers per hour.");
     for (int i = 0; i < 15; i++)
      {
        c.accelerate(10.0);
        System.out.println(c.licensePlate + " is moving at " + c.speed +
        " kilometers per hour.");
      }
```

```
    }
  }
```

Here's the output:

```
PB11R2300is moving at 0.0 kilometers per hour.
PB11R2300is moving at 10.0 kilometers per hour.
PB11R2300is moving at 20.0 kilometers per hour.
PB11R2300is moving at 30.0 kilometers per hour.
PB11R2300is moving at 40.0 kilometers per hour.
PB11R2300is moving at 50.0 kilometers per hour.
PB11R2300is moving at 60.0 kilometers per hour.
PB11R2300is moving at 70.0 kilometers per hour.
PB11R2300is moving at 80.0 kilometers per hour.
PB11R2300is moving at 90.0 kilometers per hour.
PB11R2300is moving at 100.0 kilometers per hour.
PB11R2300is moving at 110.0 kilometers per hour.
PB11R2300is moving at 120.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
```

2.1.10 Setter Methods

Setter methods, also known as mutator methods, merely set the value of a field to a value specified by the argument to the method. These methods almost always return void. One common idiom in setter methods is to use this.*name* to refer to the field and give the argument the same name as the field. For example,

```
class Car
 {
  String licensePlate; // e.g. "PB11R2300"
  double speed;        // kilometers per hour
  double maxSpeed;     // kilometers per hour
  // setter method for the license plate property
  void setLicensePlate(String licensePlate)
   {
     this.licensePlate = licensePlate;
   }
  // setter method for the maxSpeed property
  void setMaximumSpeed(double maxSpeed)
   {
    if (maxSpeed > 0)
      this.maxSpeed = maxSpeed;
```

```
    else
      this.maxSpeed = 0.0;
  }
  // accelerate to maximum speed
  // put the pedal to the metal
  void floorIt()
  {
     this.speed = this.maxSpeed;
  }
  void accelerate(double sp)
  {
     this.speed = this.speed + sp;
     if (this.speed > this.maxSpeed)
      {
        this.speed = this.maxSpeed;
      }
     if (this.speed <  0.0)
      {
        this.speed = 0.0;
       }
   }
 }
//Using Setter Methods, An Example
class CarTest5
 {
  public static void main(String args[])
   {
    Car c = new Car();
    c.setLicensePlate("New York A45 636");
    c.setMaximumSpeed(130.0);
    System.out.println(c.licensePlate + " is moving at " + c.speed +
    " kilometers per hour.");
    for (int i = 0; i < 15; i++)
     {
       c.accelerate(10.0);
       System.out.println(c.licensePlate + " is moving at " + c.speed +
       " kilometers per hour.");
      }
   }
```

```
}
```

Here's the output:

```
PB11R2300is moving at 0.0 kilometers per hour.
PB11R2300is moving at 10.0 kilometers per hour.
PB11R2300is moving at 20.0 kilometers per hour.
PB11R2300is moving at 30.0 kilometers per hour.
PB11R2300is moving at 40.0 kilometers per hour.
PB11R2300is moving at 50.0 kilometers per hour.
PB11R2300is moving at 60.0 kilometers per hour.
PB11R2300is moving at 70.0 kilometers per hour.
PB11R2300is moving at 80.0 kilometers per hour.
PB11R2300is moving at 90.0 kilometers per hour.
PB11R2300is moving at 100.0 kilometers per hour.
PB11R2300is moving at 110.0 kilometers per hour.
PB11R2300is moving at 120.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
```

2.1.11 Returning Values From Methods

It's often useful to have a method return a value to the class that called it. This is accomplished by the return keyword at the end of a method and by declaring the data type that is returned by the method at the beginning of the method. For example, the following getLicensePlate() method returns the current value of the licensePlate field in the Car class.

```
String getLicensePlate()
{
   return this.licensePlate;
}
```

A method like this that merely returns the value of an object's field or property is called a *getter* or *accessor* method. The signature String getLicensePlate() indicates that getLicensePlate() returns a value of type String and takes no arguments. Inside the method the line

return this.licensePlate;

returns the String contained in the licensePlate field to whoever called this method. It is important that the type of value returned by the return statement match the type declared in the method signature. If it does not, the compiler will complain.

Using Getter Methods, An Example

```
class CarTest6
```

```
{
  public static void main(String args[])
   {
   Car c = new Car();
     c.setLicensePlate("New York A41.5 636");
     c.setMaximumSpeed(130.0);
        System.out.println(c.getLicensePlate() + " is moving at "
     + c.getSpeed() + " kilometers per hour.");
     for (int i = 0; i < 15; i++)
      {
        c.accelerate(10.0);
        System.out.println(c.getLicensePlate() + " is moving at "
        + c.getSpeed() + " kilometers per hour.");
      }

   }
}
```

There's no longer any direct access to fields. Here's the output:

PB11R2300is moving at 0.0 kilometers per hour.
PB11R2300is moving at 10.0 kilometers per hour.
PB11R2300is moving at 20.0 kilometers per hour.
PB11R2300is moving at 30.0 kilometers per hour.
PB11R2300is moving at 40.0 kilometers per hour.
PB11R2300is moving at 50.0 kilometers per hour.
PB11R2300is moving at 60.0 kilometers per hour.
PB11R2300is moving at 70.0 kilometers per hour.
PB11R2300is moving at 80.0 kilometers per hour.
PB11R2300is moving at 90.0 kilometers per hour.
PB11R2300is moving at 100.0 kilometers per hour.
PB11R2300is moving at 110.0 kilometers per hour.
PB11R2300is moving at 120.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
PB11R2330is moving at 130.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.

2.1.12 Summary

A class is a blueprint or a template for creating different objects which defines its properties and behaviors. Java class objects exhibit the properties and behaviors defined by its class. A class can contain fields and methods to describe the behavior of an object. To instantiate an object in Java, use the keyword new followed by a call to

the class's constructor. In Java, a class is a type, similar to the built-in types such as int and boolean. In Java, no variable can ever hold an object. A variable can only hold a reference to an object. To access the fields of the class you use the . separator. Data types aren't much use unless you can do things with them. For this purpose classes have methods. Fields say what a class *is*. Methods say what a class *does*. The fields and methods of a class are collectively referred to as the *members of the class*. It's generally considered bad form to access fields directly. Instead it is considered good object oriented practice to access the fields only through methods. This allows you to change the implementation of a class without changing its interface. This also allows you to enforce constraints on the values of the fields.

2.1.13 Some Practice Questions

1. What is the relation between a class and an object?
2. What do you mean by object reference?
3. What are member variables and how are they accessed?

2.1.14 Suggested Readings

1. The Complete Reference by Herbert Scheild, Mc-Graw Hil
2. Programming with Java by E.Balagurusamy, Mc-Graw Hill
3. Java : A Beginner's Guide by Herbert Scwildt , Mc-Graw Hill
4. Introduction to Java Programming by Y.Daniel Cians, Prentice Hall
5. Object Oriented Programming in Java by G.T. Thampi
6. Java Programming by C. Xavier

Web Resources

www.tutorialspoint.com/java/
www.javapoint.com/java-tutorial
www.w3schools.in/java-tutorial/
www.programiz.com/java-programming
www.learnjavaonline.org

Lesson No. 2.2 **Author : Kanwal Preet Singh**

Constructors

2.2.1 Introduction

In the previous lesson we used setter methods to set the values of the object variables. There is another way to set these values. This is by use of constructors. Constructors are also methods that are used to set the values of object variables, but unlike setter methods they don't need to be invoked. The constructor methods are invoked themselves when an object of a class is created. In this lesson, we will also see how to define different methods with the same name using the concept of method overloading.

2.2.2 Objective

After reading this lesson you will be able to understand:

- Constructors
- Default Constructor
- Parameterized Constructors
- Method Overloading
- Constructor Overloading
- Garbage Collection
- "this" Keyword

2.2.3 Constructors

A constructor creates a new instance of the class. It initializes all the variables and does any work necessary to prepare the class to be used. In the line

Car c = new Car();

Car() is the constructor. A constructor has the same name as the class. If no constructor exists Java provides a generic one that takes no arguments, but it's better to write your own. You make a constructor by writing a method that has the same name as the class. Thus the Car constructor is called Car(). Constructors do not have return types. They do return an instance of their own class, but this is implicit, not explicit. The following method is a constructor that initializes license plate to an empty string, speed to zero and maximum speed to 120.0.

```
Car()
{
  licensePlate = "";
  speed  = 0.0;
  maxSpeed = 120.0;
}
```

We can rewrite the program written in previous lesson to include a constructor in the following way:

```
class Car
{
 String licensePlate;     // e.g. "PB11R2300"
 double speed;    // in kilometers per hour
 double maxSpeed; // in kilometers per hour
 public Car() //Constructor
  {
    licensePlate = "";
    speed  = 0.0;
    maxSpeed = 120.0;
  }
 // accelerate to maximum speed
 // put the pedal to the metal
 void floorIt()
  {
    this.speed = this.maxSpeed;
  }
 void accelerate(double sp)
  {
    this.speed = this.speed + sp;
```

```
    if (this.speed > this.maxSpeed)
     {
       this.speed = this.maxSpeed;
     }
    if (this.speed <  0.0)
     {
       this.speed = 0.0;
     }
  }
}
class CarTest4
 {
   public static void main(String[] args)
    {
     Car c = new Car();
     c.licensePlate = " PB11R2300";
     c.maxSpeed = 130.0;
     System.out.println(c.licensePlate + " is moving at " + c.speed +
     " kilometers per hour.");
     for (int i = 0; i < 15; i++)
      {
        c.accelerate(10.0);
        System.out.println(c.licensePlate + " is moving at " + c.speed +
        " kilometers per hour.");
      }
    }
  }
```

Here's the output:

PB11R2300is moving at 0.0 kilometers per hour.
PB11R2300is moving at 10.0 kilometers per hour.
PB11R2300is moving at 20.0 kilometers per hour.
PB11R2300is moving at 30.0 kilometers per hour.
PB11R2300is moving at 40.0 kilometers per hour.
PB11R2300is moving at 50.0 kilometers per hour.
PB11R2300is moving at 60.0 kilometers per hour.
PB11R2300is moving at 70.0 kilometers per hour.
PB11R2300is moving at 80.0 kilometers per hour.
PB11R2300is moving at 90.0 kilometers per hour.
PB11R2300is moving at 100.0 kilometers per hour.

PB11R2300is moving at 110.0 kilometers per hour.
PB11R2300is moving at 120.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.

The constructors have the following properties:

- Constructors are special methods.
- Constructor methods have the same name as the class itself.
- Constructors look just like methods, but they have no return type, not even void.
- Constructors are invoked only by using the *new* keyword, not the dot notation like methods.

2.2.4 Default Constructor

If you examine the first Car class code, the Car class did not have any constructors. However, you could still call "new Car()" and get a Car object. What constructor got called? The *Default* constructor is a no-arguments constructor that is provided by Java if you define a class without explicitly defining any constructors. The Default constructor allows you to create objects of classes that have no specifically designed constructors. The default constructor does not exist when the class contains any other constructor. One of the most common mistakes in Java is to rely on a Default constructor that no longer exists. Take, for example, the following class definition:

```
class Car
 {
  String licensePlate;     // e.g. "PB11R2300"
  double speed;    // in kilometers per hour
  double maxSpeed; // in kilometers per hour

  public Car(String licensePlate, double speed, double maxSpeed) //Constructor
   {
     this.licensePlate = licensePlate;
     if(maxSpeed > 0)
        this.maxSpeed = maxSpeed;
     else
        this.maxSpeed = 0.0;
     if (speed > this.maxSpeed)
        this.speed = this.maxSpeed;
     if (speed < 0)
        this.speed = 0.0;
```

```
    else
       this.speed = speed;
   }
 }
```

Given this definition of the Car class, what is wrong with the following code?

```
Car wifeCar = new Car("PB11AE2121", 60,130);
Car myCar = new Car();
myCar.licensePlate = PB11R2300;
myCar.speed = 80;
myCar.maxSpeed = 120;
```

Do you see it? Will "new Car()" compile? No, it won't, because there is no longer a constructor that takes no arguments. The Default constructor was there until you provided the additional constructor Car(String, double, double). To fix the above code, you could add the following constructor to your Car class.

Car(){}

2.2.5 Parameterized Constructors

As you may have already seen above, we can also pass arguments to the constructors. In fact a constructor without arguments is not of much use as it will initialize all the objects to the same value. Following is an example of a constructor that accepts three arguments:

```
Car(String licensePlate, double speed, double maxSpeed)
{
   this.licensePlate = licensePlate;
   if(maxSpeed > 0)
      this.maxSpeed = maxSpeed;
   else
      this.maxSpeed = 0.0;
   if (speed > this.maxSpeed)
      this.speed = this.maxSpeed;
   if (speed < 0)
      this.speed = 0.0;
   else
      this.speed = speed;
 }
```

Or perhaps you always want the initial speed to be zero, but require the maximum speed and license plate to be specified, then you can use a two argument constructor in the following way:

```
Car(String licensePlate, double maxSpeed)
```

```
 {
   this.licensePlate = licensePlate;
   this.speed  = 0.0;
   if (maxSpeed > 0)
     this.maxSpeed = maxSpeed;
   else
     this.maxSpeed = 0.0;
 }
```

Here's the complete class:

```
class Car
 {
  String licensePlate; // e.g. "PB11R2300"
  double speed;        // kilometers per hour
  double maxSpeed;     // kilometers per hour

  Car(String licensePlate, double maxSpeed)
   {
     this.licensePlate = licensePlate;
     this.speed  = 0.0;
     if (maxSpeed > 0)
        this.maxSpeed = maxSpeed;
     else
        this.maxSpeed = 0.0;
   }

 // getter (accessor) methods
  String getLicensePlate()
   {
     return this.licensePlate;
   }

  double getMaxSpeed()
   {
     return this.maxSpeed;
   }

  double getSpeed()
   {
     return this.speed;
```

```
  }
  // accelerate to maximum speed
  // put the pedal to the metal
  void floorIt()
  {
    this.speed = this.maxSpeed;
  }
   void accelerate(double deltaV)
  {
    this.speed = this.speed + deltaV;
    if (this.speed > this.maxSpeed)
     {
       this.speed = this.maxSpeed;
     }
    if (this.speed <  0.0)
     {
       this.speed = 0.0;
     }
  }
  }
```

Notice that I've taken out several things:

- the initialization of the fields
- the setter methods

The next program uses the constructor to initialize a car rather than setting the fields directly.

```
class CarTest7
 {
  public static void main(String args[])
   {
    Car c = new Car("PB11R2300", 120.5);
    System.out.println(c.getLicensePlate() + " is moving at " + c.getSpeed() +
    " kilometers per hour.");
    for (int i = 0; i < 15; i++)
     {
       c.accelerate(10.0);
       System.out.println(c.getLicensePlate() + " is moving at " + c.getSpeed()
       + " kilometers per hour.");
     }
   }
```

```
}
```

You no longer need to know about the fields licensePlate, speed and maxSpeed. All you need to know is how to construct a new car and how to print it. You may ask whether the setLicensePlate() method is still needed since it's now set in a constructor. The general answer to this question depends on the use to which the Car class is to be put. The specific question is whether a car's license plate may need to be changed after the Car object is created. Some classes may not change after they're created; or, if they do change, they'll represent a different object. The most common such class is String. You cannot change a string's data. You can only create a new String object. Such objects are called *immutable*.

2.2.6 Constraints

One of the reasons to use constructors and setter methods rather than directly accessing fields is to enforce constraints. For instance, in the Car class it's important to make sure that the speed is always less than or equal to the maximum speed and that both speed and maximum speed are greater than or equal to zero. You've already seen one example of this in the accelerate() method which will not accelerate a car past its maximum speed.

```
void accelerate(double deltaV)
 {
   this.speed = this.speed + deltaV;
   if (this.speed > this.maxSpeed)
      this.speed = this.maxSpeed;
   if (this.speed < 0.0)
      this.speed = 0.0;
 }
```

You can also insert constraints like that in the constructor. For example, this Car constructor makes sure that the maximum speed is greater than or equal to zero:

```
Car(String licensePlate, double maxSpeed)
 {
   this.licensePlate = licensePlate;
   this.speed  = 0.0;
   if (maxSpeed >= 0.0)
     this.maxSpeed = maxSpeed;
   else
     maxSpeed = 0.0;
 }
```

2.2.7 Method Overloading

Overloading is when the same method or operator can be used on many different types of data. For instance the + sign is used to add ints as well as

concatenate strings. The plus sign behaves differently depending on the type of its arguments. Therefore the plus sign is inherently overloaded. Methods can be overloaded as well. System.out.println() can print a double, a float, an int, a long or a String. You don't do anything different depending on the type of number you want the value of. Overloading takes care of it.

Method overloading results when two or more methods in the same class have the same name but different parameters. Methods with the same name must differ in their types or number of parameters. This allows the compiler to match parameters and choose the correct method when a number of choices exist. Changing just the return type is not enough to overload a method, and will be a compile-time error. They must have a different signature. When no method matching the input parameters is found, the compiler attempts to convert the input parameters to types of greater precision. A match may then be found without error. At compile time, the right implementation is chosen based on the signature of the method call. Below is an example of a class demonstrating Method Overloading

```
public class MethodOverloadDemo
{
   void sumOfParams()
    { // First Version
       System.out.println("No parameters");
    }
   void sumOfParams(int a)
    { // Second Version
       System.out.println("One parameter: " + a);
    }
    int sumOfParams(int a, int b)
    { // Third Version
       System.out.println("Two parameters: " + a + " , " + b);
       return a + b;
     }
    double sumOfParams(double a, double b)
     { // Fourth Version
       System.out.println("Two double parameters: " + a + " , " + b);
       return a + b;
     }
    public static void main(String args[])
     {
       MethodOverloadDemo moDemo = new MethodOverloadDemo();
       int intResult;
```

```
        double doubleResult;
        moDemo.sumOfParams();
        System.out.println();
        moDemo.sumOfParams(2);
        System.out.println();
        intResult = moDemo.sumOfParams(10, 20);
        System.out.println("Sum is  " + intResult);
        System.out.println();
        doubleResult = moDemo.sumOfParams(1.1, 2.2);
        System.out.println("Sum is  " + doubleResult);
        System.out.println();
    }
}
```

Output is:

No parameters
One parameter: 2
Two parameters: 10 , 20
Sum is 30
Two double parameters: 1.1 , 2.2
Sum is 3.3000000000000003

2.2.8 Constructor Overloading

Constructor Overloading is a logical extension of method overloading, A constructor is overloaded when the same constructor with different number and types of arguments initializes an object with valid initial values. In the section above, we saw several different versions of the Car constructor, one that took three arguments and one that took two arguments and one that took no arguments. We can use all of these in a single class, though here I only use two because there really aren't any good default values for licensePlate and maxSpeed. On the other hand, 0 is a perfectly reasonable default value for speed.

```
public class Car
 {
   private String licensePlate; // e.g. " PB11R2300"
   private double speed;        // kilometers per hour
   private double maxSpeed;     // kilometers per hour
   // constructors
  public Car(String licensePlate, double maxSpeed)
    {
     this.licensePlate = licensePlate;
     this.speed = 0.0;
```

```
    if (maxSpeed >= 0.0)
      {
        this.maxSpeed = maxSpeed;
      }
    else
      {
        maxSpeed = 0.0;
      }
  }
  public Car(String licensePlate, double speed, double maxSpeed)
   {
     this.licensePlate = licensePlate;
     if (maxSpeed >= 0.0)
       {
         this.maxSpeed = maxSpeed;
       }
   else
       {
         maxSpeed = 0.0;
       }
   if (speed < 0.0)
       {
         speed = 0.0;
       }
   if (speed <= maxSpeed)
       {
         this.speed = speed;
       }
   else
       {
         this.speed = maxSpeed;
       }
   }
 // other methods...
}
```

The signature of the first constructor in the above program is Car(String, double). The signature of the second constructor is Car(String, double, double). Thus the first version of the Car() constructor is called when there is one String argument followed by one double argument and the second version is used when there is one

String argument followed by two double arguments. If there are no arguments to the constructor or two or three arguments that aren't the right type in the right order, then the compiler generates an error because it doesn't have a method whose signature matches the requested method call. For example

Error: Method Car(double) not found in class Car.

this in constructors

It is often the case that overloaded methods are essentially the same except that one supplies default values for some of the arguments. In this case, your code will be easier to read and maintain (though perhaps *marginally* slower) if you put all your logic in the method that takes the most arguments and simply invoke that method from all its overloaded variants that merely fill in appropriate default values. This technique should also be used when one method needs to convert from one type to another. For instance one variant can convert a String to an int, then invoke the variant that takes the int as an argument. This is straight-forward for regular methods, but doesn't quite work for constructors because you can't simply write a method like this:

```
public Car(String licensePlate, double maxSpeed)
 {
   Car(licensePlate, 0.0, maxSpeed);
 }
```

Instead, to invoke another constructor in the same class from a constructor you use the keyword this like so:

```
public Car(String licensePlate, double maxSpeed)
 {
    this(licensePlate, 0.0, maxSpeed);
 }
```

Must this be the first line of the constructor?

For example,

```
public class Car
{
  private String licensePlate; // e.g. "PB11R2300"
  private double speed;        // kilometers per hour
  private double maxSpeed;     // kilometers per hour
  // constructors
  public Car(String licensePlate, double maxSpeed)
   {
     this(licensePlate, 0.0, maxSpeed);
   }
 public Car(String licensePlate, double speed, double maxSpeed)
  {
```

```
    this.licensePlate = licensePlate;
    if (maxSpeed >= 0.0)
     {
       this.maxSpeed = maxSpeed;
     }
    else
     {
       maxSpeed = 0.0;
     }
    if (speed < 0.0)
     {
       speed = 0.0;
     }
    if (speed <= maxSpeed)
     {
       this.speed = speed;
     }
    else
     {
       this.speed = maxSpeed;
     }
  }

  // other methods...
}
```

This approach saves several lines of code. In also means that if you later need to change the constraints or other aspects of construction of cars, you only need to modify one method rather than two. This is not only easier; it gives bugs fewer opportunities to be introduced either through inconsistent modification of multiple methods or by changing one method but not others.

2.2.9 Garbage Collection

So far, this section has been about creating objects. What about destroying them? In Java, the destruction of objects takes place automatically. An object exists in the heap, and it can be accessed only through variables that hold references to the object. What should be done with an object if there are no variables that refer to it? Such things can happen. Consider the following two statements (though in reality, you'd never do anything like this):

```
Student std = new Student("Hardeep");
std = null;
```

In the first line, a reference to a newly created Student object is stored in the variable std. But in the next line, the value of std is changed and the reference to the Student object is gone. In fact, there are now no references whatsoever to that object stored in any variable. So there is no way for the program ever to use the object again. It might as well not exist. In fact, the memory occupied by the object should be reclaimed to be used for another purpose.

Java uses a procedure called **garbage collection** to reclaim memory occupied by objects that are no longer accessible to a program. It is the responsibility of the system, not the programmer, to keep track of which objects are "garbage." In the above example, it was very easy to see that the Student object had become garbage. Usually, it's much harder. If an object has been used for a while, there might be several references to the object stored in several variables. The object doesn't become garbage until all those references have been dropped.

In many other programming languages, it's the programmer's responsibility to delete the garbage. Unfortunately, keeping track of memory usage is very error-prone, and many serious program bugs are caused by such errors. A programmer might accidently delete an object even though there are still references to that object. This is called a dangling pointer error and it leads to problems when the program tries to access an object that is no longer there. Another type of error is a memory leak, where a programmer neglects to delete objects that are no longer in use. This can lead to filling memory with objects that are completely inaccessible and the program might run out of memory even though, in fact, large amounts of memory are being wasted.

Because Java uses garbage collection, such errors are simply impossible. Garbage collection is an old idea and has been used in some programming languages since the 1960s. You might wonder why all languages don't use garbage collection. In the past, it was considered too slow and wasteful. However, research into garbage collection techniques combined with the incredible speed of modern computers have combined to make garbage collection feasible. Programmers should rejoice.

2.2.10 “this” Keyword

All instance methods have automatic access to other instance methods and any data (instance variables) defined for the object. In the example below, the pimpMyRide method calls the getDescription() method and uses Car instance variables.

```
public class Car
  {
    String color;
    String type;
       {
         color="red";
         type="sedan";
```

```
    }
  String getDescription()
   {
     String desc = "This is a " + color + " " + type;
     return desc;
   }
  void pimpMyRide(String newColor, String customized)
   {
    color = newColor;
    type = customized + " " + type;
    System.out.println(getDescription());
   }
}
```

Within an instance method or a constructor, the keyword *this* is a reference to the current object. In other words, *this* refers to the current instance. You can use *this* to refer to any instance variable or instance method of the object from within an instance method or a constructor. Rewriting the code above should help demonstrate *this*.

```
public class Car {
  String color;
  String type;
    {
    this.color="red";
    this.type="sedan";
  }
  String getDescription(){
    String desc = "This is a " + this.color + " " + this.type;
    return desc;
  }
  void pimpMyRide(String newColor, String customized) {
    this.color = newColor;
    this.type = customized + " " + this.type;
    System.out.println(this.getDescription());
  }
}
```

Even though this code works, the use of *this* here doesn't seem to make much sense? In fact, it might just make things more complex. However, consider the pimpMyRide method if it were written as shown below.

```
  void pimpMyRide(String color, String type) {
     color = color;              //??? Which color is which
```

```
    type = type + " " + type;     //??? Which type is which
    System.out.println(getDescription());
  }
```

The parameters passed into pimpMyRide now conflict with the instance variable names. This code will compile, but it won't run correctly, as will be discussed. When parameters passed into a constructor or method have the same name as instance variables, this is called *shadowing a field.* Shadowing helps clarify how the parameter will be used to set or modify an instance variable. To fix the problem, the *this* keyword helps to disambiguate what is the object's instance variable and what is just the parameter.

```
    void pimpMyRide(String color, String type) {
  this.color = color;
  this.type = type + " " + this.type;
  System.out.println(getDescription());
  }
```

Static methods do not have access to *this* because when you enter a static method, you are not in an object instance. "this" only applies to instances. In the above case, *this* helps provide clarity, but if alternative parameter names are used, you can avoid having to use *this*. In another case, the *this* keyword is the only way to accomplish the task. Consider the Car example with two constructors as specified below.

```
Car(){
 carCount++;
 serialNumber = carCount;
}
Car(String c, String t){
 carCount++;
 serialNumber = carCount;
 color = c;
 type = t;
}
```

Do you notice some similarities between the two constructors? What happens if the way serial numbers are handled is changed? In this case, you would have to modify two constructors where duplicate code is used to deal with serial numbers. This is not a very good reuse design! How can the common code be isolated and reused in these constructors? Constructors can call other constructors in the same class using *this*. Using *this* is similar to calling a method from within your constructor: you must match the appropriate argument list. To call another constructor, use *this* on the **first line** of another constructor.

```
  Car(){
   carCount++;
   serialNumber = carCount;
 }

 Car(String c, String t){
   this();
   color = c;
   type = t;
 }
```

By having the second constructor call the first one, you don't need to repeat code. Any change made to the first constructor will also affect the second constructor.
Setting up one constructor to call another is called constructor chaining.
Using the *this* keyword is awkward at first, but there are only two uses for it:

- Referring to the current instance (e.g., this.color or) this.getDescription()
- Referring to another constructor (e.g., this(<arg>) from inside a constructor.

)

2.2.11 Summary

Constructors are methods that are used to set the values of object variables, but unlike setter methods they don't need to be invoked. The *Default* constructor is a no-arguments constructor that is provided by Java if you define a class without explicitly defining any constructors. The Default constructor allows you to create objects of classes that have no specifically designed constructors. We can also pass arguments to constructors in the same way as to ordinary methods. Method overloading results when two or more methods in the same class have the same name but different parameters. Methods with the same name must differ in their types or number of parameters. Constructor Overloading is an extension of method overloading, A constructor is overloaded when the same constructor with different number and types of arguments initializes an object with valid initial values. In Java, the destruction of objects takes place automatically. Java uses a procedure called garbage collection to reclaim memory occupied by objects that are no longer accessible to a program. It is the responsibility of the system, not the programmer, to keep track of which objects are "garbage." Within an instance method or a constructor, the keyword *this* is a reference to the current object. In other words, *this* refers to the current instance. You can use *this* to refer to any instance variable or instance method of the object from within an instance method or a constructor.

2.2.12 Some Practice Questions

1. What are constructors? Explain different properties of constructors.
2. What happens when you don't define a constructor in a class?
3. Explain method overloading and constructor overloading in detail?

2.2.13 Suggested Readings

1. The Complete Reference by Herbert Scheild, Mc-Graw Hill
2. Programming with Java by E.Balagurusamy, Mc-Graw Hill
3. Java : A Beginner's Guide by Herbert Schildt, Mc-Graw Hill
4. Introduction to Java Programming by Y.Daniel Cians, Prentice Hall
5. Object Oriented Programming in Java by G.T. Thampi
6. Java Programming by C. Xavier

Web Resources

www.tutorialspoint.com/java/
www.javapoint.com/java-tutorial
www.w3schools.in/java-tutorial/
www.programiz.com/java-programming
www.learnjavaonline.org

BCA Part-III

Lesson No. 2.3 **Author : Kanwal Preet Singh**

Passing Objects and Access Specifiers

2.3.1 Introduction

We can pass objects as arguments to methods just as other variables. Similarly a method can return s to the object to the calling method. In this lesson, we will see how to pass objects as arguments and how to return objects to the calling method. We will also discuss the concept of recursion in which a method calls itself again and again. There are four access specifiers used in Java – public, default, protected and private which restrict the accessibility of the variables to different parts of the program. In the end we will discuss static members which are those members that belong to the class rather than to the object.

2.3.2 Objective

After reading this lesson you will be able to understand:

- How to pass objects as arguments
- How to return objects
- Recursion
- Access Specifiers
- Static variables and methods

2.3.3 Passing Objects as Arguments

So far we have only been passing simple types as parameters to methods. However, we can also pass objects to methods. Consider the following example:

```
// Objects may be passed to methods.
class Test
 {
   int a, b;
   Test(int i, int j)
    {
      a = i;
      b = j;
    }
  // return true if o is equal to the invoking object
  boolean equals(Test o)
   {
     if(o.a == a && o.b == b)
       return true;
     else
       return false;
   }
 }
class PassOb
 {
  public static void main(String args[])
   {
     Test ob1 = new Test(105, 20);
     Test ob2 = new Test(105, 20);
     Test ob3 = new Test(-10, -10);

     System.out.println("ob1 == ob2: " + ob1.equals(ob2));
     System.out.println("ob1 == ob3: " + ob1.equals(ob3));
   }
}
```

The output is:

ob1 == ob2: true
ob1 == ob3: false

In the above example, we pass an object of class Test to the method equals(). The method compares the class variables of the object that invoked it with object that has been passed to the method and returns true if they are equal.

There are two ways that a computer language can pass an argument to a subroutine. The first way is call-by-value. This method copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument use to call it. The second an argument can be passed is call-by-reference. In this method a reference to an argument is passed to the parameter. Inside the subroutine this reference is used to assist the actual argument specified in the call i.e. changes made to the parameter will affect the argument used to call the subroutine. Java uses both methods, depending upon what is passed.

In Java, when you pass a simple type to a method, it is passed by value. When you pass an object to a method, it is passed by a reference. When we create a variable of a class type, we are only creating a reference to an object. Thus, when we pass this reference to a method the parameter that receives it will refer to the same object as that referred to by the argument which means that objects are passed to methods by use of call-by-reference. This will be clear from the following example:

```
// Simple Types are passed by value. Objects are passed by reference.
class Test
 {
   int a, b;
   Test(int i, int j)
     {
      a = i;
      b = j;
     }
    //Pass by value
   void meth(int i, int j)
     {
       i *= 2;
       j /= 2;
     }
   // pass an object
   void meth(Test o)
    {
      o.a *=  2;
      o.b /= 2;
    }
 }
class CallByRef
  {
```

```
  public static void main(String args[])
   {
      int  a=5, b=10;
      Test ob = new Test(5, 10);
     System.out.println("a and b before call: " + a + " " + b);
     ob.meth(a, b);
     System.out.println("a and b after call: " + a + " " + b);

     System.out.println("ob.a and ob.b before call: " +ob.a + " " + ob.b);
     ob.meth(ob);
     System.out.println("ob.a and ob.b after call: " + ob.a + " " + ob.b);
   }
 }
```

The output is:

a and b before call: 5 10
a and b after call: 5 10
ob.a and ob.b before call: 5 10
ob.a and ob.b after call: 10 5

It is clear from the above example that when we pass a simple type to a method it is passed by value and when we pass an object to a method it is passed by reference. In fact when an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed referred to n object, the copy of that value will still refer to the same object that its corresponding argument does.

2.3.4 Returning objects

A method can return any type of data including objects that we create. In the following program, the incrByTen() method returns an object in which the value of a is ten greater than it is in the invoking object.

```
// Returning an object.
class Test
 {
  int a;
  Test(int i)
   {
     a = i;
   }
  Test incrByTen()
   {
     Test temp = new Test(a+10);
```

```
    return temp;
   }
  }
class RetOb
 {
  public static void main(String args[])
   {
    Test ob1 = new Test(2);
    Test ob2;
    ob2 = ob1.incrByTen();
    System.out.println("ob1.a: " + ob1.a);
    System.out.println("ob2.a: " + ob2.a);
    ob2 = ob2.incrByTen();
    System.out.println("ob2.a after second increase: " + ob2.a);
   }
 }
```

2.3.5 Recursion

The term recursion generally refers to the technique of repeatedly splitting a task into "the same task on a smaller scale". In practice, it often means making a method that calls itself. A method called in this way is often called a recursive method. Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive.

The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N. for example, 3 factorial is 1×2×3, or 6. Here is how a factorial can be computed by use of a recursive method.

```
class Factorial
  {
    int fact(int n)
     {
       int result;
       if ( n ==1) return 1;
       result = fact (n-1) * n;
       return result;
     }
  }
class Recursion
  {
```

```
    public static void main (String args[])
     {
        Factorial f =new Factorial();
        System.out.println("Factorial of 3 is " + f.fact(3));
        System.out.println("Factorial of 4 is " + f.fact(4));
        System.out.println("Factorial of 5 is " + f.fact(5));
     }
  }
```

The output from this program is shown here:

Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120

If you are unfamiliar with recursive methods, then the operation of **fact()** may seem a bit confusing. Here is how it works. When **fact()** is called with an argument of 1, the function returns 1; otherwise it returns the product of **fact(n-1)*n.** to evaluate this expression, **fact()** is called with **n-1**. this process repeats until **n** equals 1 and the calls to the method begin returning.

To better understand how the **fact()** method works, let's go through a short example. When you compute the factorial of 3, the first call to **fact()** will cause a second call to be made with an argument of 2. this invocation will cause **fact()** to be called a third time with an argument of 2. This call will return 1, which is then be called a third time with an argument of 1. This call will return1, which is then multiplied by 2 (the value of **n** in the second invocation). This result (which is 2) is then returned to the original invocation of **fact()** and multiply by 3 (the original value of **n).** This yields the answer, 6. You might find it interesting to insert **println()** statements into **fact()** which will show at what level each call is and what the intermediate answers are. The following diagram shows how it works:

```
fact(3)
   fact(2)
      fact(1)
      return 1
   return 2*1 = 2
return 3*2 = 6
```

When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start. A recursive call does not make a new copy of the method. Only the arguments are new. As each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the call inside the method. Recursive methods could be said to "telescope" out and back.

Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional function calls. Many recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables, it is possible that the stack could be exhausted. If this occurs, the java run-time system will cause an exception. However, you probably will not have to worry about this unless a recursive routine runs wild.

Our factorial implementation exhibits the two main components that are required for every recursive function. The *base case* returns a value without making any subsequent recursive calls. It does this for one or more special input values for which the function can be evaluated without recursion. For fact(), the base case is N = 1. The *reduction step* is the central part of a recursive function. It relates the function at one (or more) inputs to the function evaluated at one (or more) other inputs. For fact(), the reduction step is N * fact(N-1). All recursive functions must have these two components. Furthermore, the sequence of parameter values must *converge* to the base case. For fact(), the value of N decreases by one for each call, so the sequence of parameter values converges to the base case N = 1.

The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives. For example, the QuickSort sorting algorithm is quite difficult to implement in an iterative way.

2.3.6 Java Access Specifiers

access to classes, constructors, methods and fields are regulated using access modifiers i.e. a class can control what information or data can be accessible by other classes. To take advantage of encapsulation, you should minimize access whenever possible. Java provides a number of access modifiers to help you set the level of access you want for classes as well as the fields, methods and constructors in your classes. A member has package or default accessibility when no accessibility modifier is specified. The following Access Modifiers are provided in Java:

- **private**
- **protected**
- **default**
- **public**

public access modifier

Fields, methods and constructors declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package.

private access modifier

The private (most restrictive) fields or methods cannot be used for classes and Interfaces. It also cannot be used for fields and methods within an interface. Fields, methods or constructors declared private are strictly controlled, which means they

cannot be accessed by anywhere outside the enclosing class. A standard design strategy is to make all fields private and provide public getter methods for them.

protected access modifier

The protected fields or methods cannot be used for classes and Interfaces. It also cannot be used for fields and methods within an interface. Fields, methods and constructors declared protected in a superclass can be accessed only by subclasses in other packages. Classes in the same package can also access protected fields, methods and constructors as well, even if they are not a subclass of the protected member's class.

default access modifier

Java provides a default specifier which is used when no access modifier is present. Any class, field, method or constructor that has no declared access modifier is accessible only by classes in the same package. The default modifier is not used for fields and methods within an interface.

Class Access Control Modifiers

1. A class can have either the **public** or the **default** access control level.
2. You make a class public by using the public access control modifier.
3. A class whose declaration bears no access control modifier has default access.

Understand the effects of public and private access

```
class Test
 {
   int a; // default access
   public int b; // public access
   private int c; // private access
   void setc(int i)
    {
      c = i;
    }

   int getc()
    {
      return c;
    }
 }
class AccessTest
 {
   public static void main(String args[])
     {
       Test ob = new Test();
```

```
        ob.a = 10;
        ob.b = 20;
        // ob.c = 100; Not Possible. Can't access private member outside its class
        ob.setc(100);
        System.out.println("a, b, and c: " + ob.a + " " + ob.b + " " + ob.getc());
    }
}
```

We will come back to the access specifiers when we have done packages. We can understand default and protected access specifiers only when we have full understanding of packages.

2.3.7 Static Variables

Adding a couple of additional fields to the car example should help to clarify the class variable concept. First, each car has a serialNumber (called a vehicle identification number or VIN). This is pretty simple. Based on what you have already learned, simply add a serialNumber instance variable to the Car class.

```
public class Car
{
    String color;
    String type;
    int serialNumber;
    ...  // The rest of the class goes here. This symbol (...) is used
         // throughout this text to indicate that not all the code is
         // is shown but only the code that is pertinent at the time.
}
```

This serialNumber, however, must be a unique integer for each car created. In this case, the serialNumber should be unique for each car object created from the Car class. To make sure each car has a unique serialNumber, you can use the Car class and a class variable to keep track of the total number of cars created.

```
public class Car
{
   String color;
   String type;
   int serialNumber;
   static int carCount;
      ...
}
```

Here the modifier "**static**" was added to the carCount variable in this class. The carCount variable is related to the class, not any single Car object. To access this

variable, you use the name of the class and the class variable name. For example, the line of code below sets the carCount to 1.

```
Car.carCount = 1;
```

Oddly, you can use either the class name (Car) or any object reference of that type to access class variables, so the following code would do the same thing:

```
Car myCar = new Car("black", "Ranger");
myCar.carCount = 1;
```

This makes it look like carCount is an instance variable, doesn't it? Therefore, it is considered clearer and preferred, to use the class name when accessing class/static variables.

Note that the carCount variable is created and initialized not when an object is instantiated but when the class is first loaded into the JVM by the class loader! That is because the carCount data is associated with the class (Car) and not any single instance of the class (myCar). No matter how many Cars get created, there is still just one carCount. Recall that what is needed is to set the serialNumber for every Car and that number must be unique. The carCount class variable can be used to achieve this goal. However, to pull this off, more work on the constructors for the Car class is needed.

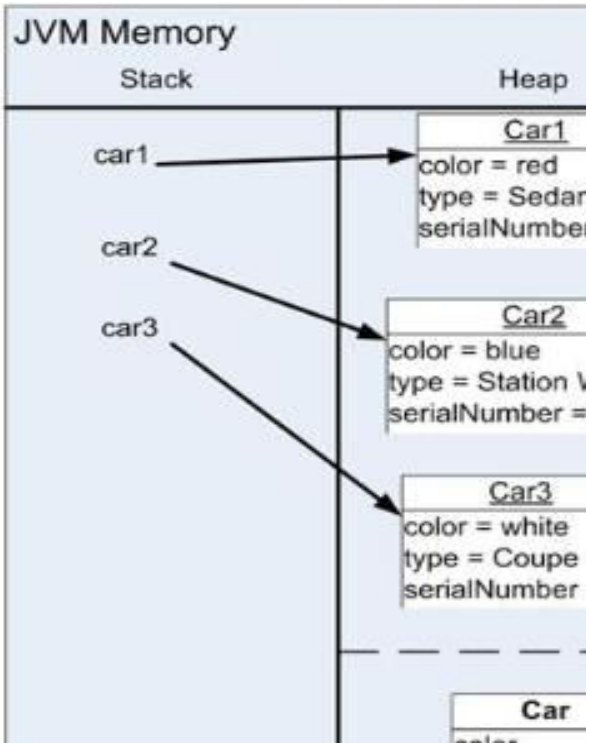
```
Car()
  {
     carCount++;
     serialNumber = carCount;
  }
Car(String c, String t)
  {
       color = c;
       type = t;
       carCount++;
       serialNumber = carCount;
  }
```

This code increments the number of Cars (carCount) every time a Car is created and then assigns carCount to the car object's serialNumber. Now each Car has a unique serial number, created from the class's carCount. When each car is created, the constructor increases the value of carCount by one. The carCount variable represents the total number of cars that have been created so far. Now take a look at the code below to visualize what is happening.

```
Car car1, car2, car3;        //create 3 object references.
car1 = new Car("red","Sedan");
car2 = new Car("blue", "Station Wagon");
car3 = new Car("white", "Coupe");
```

When the declaration of a Car reference is made (Car car1), the class loader must load the Car class into the JVM. At that time, carCount is initialized and available. Every object type used in an application has a respective Class object stored in a special place on the JVM's heap. This Class object contains all of the details about the object type:

- What properties it has
- What the properties' types are
- What methods the class has and what arguments the methods take
- What code is executed when a method is called and so forth. Static data is stored in the Class object on the JVM's heap.



All three cars have their own independent color, type, and serialNumber, but they share one car Count variable.

Class variables are global (only one exists in memory per class). All instances share the one carCount, which is obtained through the class. Each instance has its own color, type and serialNumber. All code in the JVM can reference static information of a class depending on the access modifier.

2.3.8 Static Method

The static keyword can also be applied to a method. A static method, like a static variable, is associated with the class, not the objects (instances). Static methods

are also called class methods (versus nonstatic methods, which are instance methods). Methods you have seen so far are instance methods. Below, another instance method, drive(), is added to the Car class.

```
public class Car
{
  String color;
  String type;

  void drive()
   {
     System.out.println("Put 'er in gear and drive it like you stole it");
   }
    ...
}
```

To call an instance (nonstatic) method, you must have an object reference.

```
Car c = new Car();
c.drive();          //Correct
Car.drive();  //Wrong (what car are you driving?)
```

To define a static method, simply add the *static* keyword as a modifier to the method as shown below.

```
public class Car
 {
    ...
    static void resetCarCount()
     {
        carCount = 0;
     }
   ...
 }
```

To call a static method, you only need the name of the class.

```
Car c = new Car();
c.resetCarCount();    //Legal but confusing
Car.resetCarCount();  //Proper way to code
```

As shown above, just as with class variables, you can call on a static method using any object of that type. However, this makes it look like resetCarCount() is an instance method, doesn't it? Again, it is considered clearer and preferred, to use the class name when accessing static methods.

Static methods do not have access to object data. Looking at the example below, what color are you changing if you called resetCarCount?

```
public class Car
 {
    ...
    static void resetCarCount()
     {
        carCount = 0;
        color = "blue"; //Wrong - which instance's color are you changing?
     }
   ...
 }
```

Now you might be asking yourself, "What's the point of static methods?" If so, that's a good and fair question. Static methods essentially have two purposes.

1) They are used to access (update or fetch) class variable data. Although this can be done with any instance of the class, it is considered more appropriate to use class methods for this purpose. In some cases, you may not have an instance of an object created before the data is needed. You don't want to have to create an object just to be able to access class variables.
2) Static methods provide functionality without the need for an object/instance. For example, mathematical formulas are great reasons to have static methods. Should you have to create an instance of some object to compute sine, cosine or tangent? Examine the Math class to see some excellent uses of static methods. There is no need to create an instance of Math to compute the absolute value of a number!

2.3.9 Static Initialization Block

Classes can also have initialization blocks. More precisely, they can have *static initialization* blocks. Like a normal initialization block, a static initialization block is a normal block of code enclosed in braces {} preceded by the *static* keyword.

```
public class Car
 {
      ...
      static
       {
          carCount = 1;
       }
      ...
 }
```

A class can have any number of static initialization blocks and they can appear anywhere in the class. They are often grouped together for better maintenance. The static initialization blocks are called in the order they appear in the code. Static

initialization code blocks get executed once when the class is loaded. Just like static methods, they can only initialize static variables of the class.

2.3.10 Summary

We can pass objects as arguments to methods just as other variables. Similarly a method can return s to the object to the calling method. The term recursion generally refers to the technique of repeatedly splitting a task into "the same task on a smaller scale". In practice, it often means making a method that calls itself. A method called in this way is often called a recursive method. The access to classes, constructors, methods and fields are regulated using access modifiers i.e. a class can control what information or data can be accessible by other classes. There are four access specifiers used in Java – public, default, protected and private. Static members are those members that belong to the class rather than to the object. There exists only one copy of static variables for all the class objects and static methods can access only the static variables and other static methods.

2.3.11 Short Answer Type Questions

1. Explain the concept of recursion?
2. Write a program to print Fibonacci series using recursion?
3. Why do we use static variables?

2.3.12 Long Answer Type Questions

1. Explain in detail with the help of an example how to pass objects as arguments and how to return objects to the calling method in java.
2. Explain in detail the different access specifiers used in Java.

2.3.13 Suggested Readings

1. The Complete Reference by Herbert Scheild, Mc-Graw Hill
2. Programming with Java by E.Balagurusamy, Mc-Graw Hill
3. Java : A Beginner's Guide by Herbert Schildt, Mc-Graw Hill
4. Introduction to Java Programming by Y.Daniel Cians, Prentice Hall
5. Object Oriented Programming in Java by G.T. Thampi
6. Java Programming by C. Xavier

Web Resources

www.tutorialspoint.com/java/
www.javapoint.com/java-tutorial
www.w3schools.in/java-tutorial/
www.programiz.com/java-programming
www.learnjavaonline.org

Nested and Inner Classes

2.4.1 Introduction

We can define a class within another class. Such classes are called nested classes. Nested classes are divided into two categories: static and non-static. There are several compelling reasons for using nested classes and they are discussed in this chapter. The other topics that we will discuss in this lesson are strings and inheritance. String is a special class built into the Java language defined in the java.lang package. The String class represents character strings. String literals in Java programs, such as "abc", are implemented as instances of this class.

2.4.2 Objective

After reading this lesson you will be able to understand:

- Nested Classes
- Static Nested Classes
- Inner Classes
- Strings
- Java String Functions

2.4.3 Nested Classes

The Java programming language allows you to define a class within another class. Such a class is called a *nested class* and is illustrated here:

```
class OuterClass
{
   ...
   class NestedClass
```

```
    {
        ...
    }
}
```

Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are simply called *static nested classes*. Non-static nested classes are called *inner classes*.

```
class OuterClass
{
    ...
    static class StaticNestedClass
    {
        ...
    }
    class InnerClass
    {
        ...
    }
}
```

A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class. As a member of the OuterClass, a nested class can be declared private, public, protected, or *package private*. (Recall that outer classes can only be declared public or *package private*.)

Why Use Nested Classes?

There are several compelling reasons for using nested classes, among them:

- It is a way of logically grouping classes that are only used in one place.
- It increases encapsulation.
- Nested classes can lead to more readable and maintainable code.

Logical grouping of classes—If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.

Increased encapsulation—Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world. More readable, maintainable code—Nesting small classes within top-level classes places the code closer to where it is used.

2.4.4 Static Nested Classes

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class — it can use them only through an object reference.

Note: A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

Static nested classes are accessed using the enclosing class name:

OuterClass.StaticNestedClass

For example, to create an object for the static nested class, use this syntax:

OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();

The definition of a static nested class looks just like the definition of any other class, except that it is nested inside another class and it has the modifier static as part of its declaration. A static nested class is part of the static structure of the containing class. It can be used inside that class to create objects in the usual way. If it has not been declared private, then it can also be used outside the containing class, but when it is used outside the class, its name must indicate its membership in the containing class. This is similar to other static components of a class: A static nested class is part of the class itself in the same way that static member variables are parts of the class itself.

For example, suppose a class named WireFrameModel represents a set of lines in three-dimensional space. (Such models are used to represent three-dimensional objects in graphics programs.) Suppose that the WireFrameModel class contains a static nested class, Line, that represents a single line. Then, outside of the class WireFrameModel, the Line class would be referred to as WireFrameModel.Line. Of course, this just follows the normal naming convention for static members of a class. The definition of the WireFrameModel class with its nested Line class would look, in outline, like this:

```
public class WireFrameModel
 {
   . . . // other members of the WireFrameModel class

   static public class Line
    {
       // Represents a line from the point (x1,y1,z1)
       // to the point (x2,y2,z2) in 3-dimensional space.
     double x1, y1, z1;
```

```
    double x2, y2, z2;
  } // end class Line
  . . . // other members of the WireFrameModel class
} // end WireFrameModel
```

Inside the WireFrameModel class, a Line object would be created with the constructor "new Line()". Outside the class, "new WireFrameModel.Line()" would be used. A static nested class has full access to the static members of the containing class, even to the private members. Similarly, the containing class has full access to the members of the nested class. This can be another motivation for declaring a nested class, since it lets you give one class access to the private members of another class without making those members generally available to other classes.

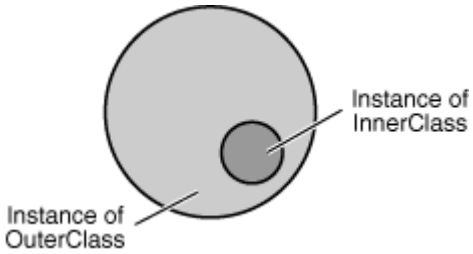
When you compile the above class definition, two class files will be created. Even though the definition of Line is nested inside WireFrameModel, the compiled Line class is stored in a separate file. The name of the class file for Line will be WireFrameModel$Line.class.

2.4.5 Inner Classes

As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself. Objects that are instances of an inner class exist *within* an instance of the outer class. Consider the following classes:

```
class OuterClass
{
   ...
   class InnerClass
    {
       ...
    }
}
```

An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance. The next figure illustrates this idea.



An Instance of InnerClass Exists Within an Instance of OuterClass. To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

OuterClass.InnerClass innerObject = outerObject.new InnerClass();

In Java, a nested class is any class whose definition is inside the definition of another class. Nested classes can be either named or anonymous. A named nested class, like most other things that occur in classes, can be either static or non-static.

Non-static nested classes are referred to as inner classes. Inner classes are not, in practice, very different from static nested classes, but a non-static nested class is actually associated with an object rather than to the class in which it is nested. This can take some getting used to.

Any non-static member of a class is not really part of the class itself (although its source code is contained in the class definition). This is true for inner classes, just as it is for any other non-static part of a class. The non-static members of a class specify what will be contained in objects that are created from that class. The same is true -- at least logically -- for inner classes. It's as if each object that belongs to the containing class has its own copy of the nested class. This copy has access to all the instance methods and instance variables of the object, even to those that are declared private. The two copies of the inner class in two different objects differ because the instance variables and methods they refer to are in different objects. In fact, the rule for deciding whether a nested class should be static or non-static is simple: If the nested class needs to use any instance variable or instance method from the containing class, make the nested class non-static. Otherwise, it might as well be static.

From outside the containing class, a non-static nested class has to be referred to using a name of the form variableName.NestedClassName, where variableName is a variable that refers to the object that contains the class. This is actually rather rare, however. A non-static nested class is generally used only inside the class in which it is nested, and there it can be referred to by its simple name.

In order to create an object that belongs to an inner class, you must first have an object that belongs to the containing class. (When working inside the class, the object "this" is used implicitly.) The inner class object is permanently associated with the containing class object, and it has complete access to the members of the containing class object. Looking at an example will help, and will hopefully convince you that inner classes are really very natural. Consider a class that represents poker games. This class might include a nested class to represent the players of the game. This structure of the PokerGame class could be:

```
public class PokerGame {  // Represents a game of poker.
  private class Player {  // Represents one of the players in this game.
     .
```

```
    .
    .
  } // end class Player
  private Deck deck;      // A deck of cards for playing the game.
  private int pot;        // The amount of money that has been bet.
  .
  .
  .
} // end class PokerGame
```

If game is a variable of type PokerGame, then, conceptually, game contains its own copy of the Player class. In an instance method of a PokerGame object, a new Player object would be created by saying "new Player()", just as for any other class. (A Player object could be created outside the PokerGame class with an expression such as "game.new Player()". Again, however, this is very rare.) The Player object will have access to the deck and pot instance variables in the PokerGame object. Each PokerGame object has its own deck and pot and Players. Players of that poker game use the deck and pot for that game; players of another poker game use the other game's deck and pot. That's the effect of making the Player class non-static. This is the most natural way for players to behave. A Player object represents a player of one particular poker game. If Player were a static nested class, on the other hand, it would represent the general idea of a poker player, independent of a particular poker game.

2.4.6 Strings

String is a special class built into the Java language defined in the java.lang package. The String class represents character strings. String literals in Java programs, such as "abc", are implemented as instances of this class. Strings are immutable; that is, they cannot be modified once created. For example:

```
String str = "This is string literal";
```

On the right hand side a String object is created represented by the string literal. Its object reference is assigned to the str variable. The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings. For example:

```
String str = "First part" + " second part";
// --- Is the same as:
String str = "First part second part";
```

Integers will also be converted to String after the (+) operator:

```
String str = "Age=" + 25;
```

Each Java object has the String toString() inherited from the Object class. This method provides a way to convert objects into Strings. Most classes override the default behavior to provide more specific (and more useful) data in the returned String. The

String class provides a nice set of methods for string manipulation. Since String objects are immutable, all methods return a new String object. For example:

name = name.trim();

The trim() method returns a copy of the string with leading and trailing whitespace removed. Note that the following would do nothing useful:

name.trim(); // wrong!

This would create a new trimmed string and then throw it away.

If 2 or more Strings have the same set of characters in the same sequence then they share the same reference in memory. Below illustrates this phenomenon.

String str1 = “My name is bob”;
String str2 = “My name is bob”;
String str3 = “My name ”+ “is bob”; //Compile time expression
String name = “bob”;
String str4 = “My name is” + name;
String str5 = new String(“My name is bob”);

In the above code all the String references str1, str2 and str3 denote the same String object, initialized with the character string: “My name is bob”. But the Strings str4 and str5 denote new String objects.

```
//String Equality
public class StringsDemo1
 {
   public static void main(String[] args)
    {
        String str1 = "My name is bob";
        String str2 = "My name is bob";
        String str3 = "My name " + "is bob"; //Compile time expression
        String name = "bob";
        String str4 = "My name is " + name;
        String str5 = new String("My name is bob");
        System.out.println("str1 == str2 : " + (str1 == str2));
        System.out.println("str2 == str3 : " + (str2 == str3));
        System.out.println("str3 == str1 : " + (str3 == str1));
        System.out.println("str4 == str5 : " + (str4 == str5));
        System.out.println("str1 == str4 : " + (str1 == str4));
        System.out.println("str1 == str5 : " + (str1 == str5));
        System.out.println("str1.equals(str2) : " + str1.equals(str2));
        System.out.println("str2.equals(str3) : " + str2.equals(str3));
        System.out.println("str3.equals(str1) : " + str3.equals(str1));
        System.out.println("str4.equals(str5) : " + str4.equals(str5));
```

```
        System.out.println("str1.equals(str4) : " + str1.equals(str4));
        System.out.println("str1.equals(str5) : " + str1.equals(str5));
    }
  }
```

Output is:

```
str1 == str2 : true
str2 == str3 : true
str3 == str1 : true
str4 == str5 : false
str1 == str4 : false
str1 == str5 : false
str1.equals(str2) : true
str2.equals(str3) : true
str3.equals(str1) : true
str4.equals(str5) : true
str1.equals(str4) : true
str1.equals(str5) : true
```

The == operator is used when we have to compare the String object references. If two String variables point to the same object in memory, the comparison returns true. Otherwise, the comparison returns false. Note that the '==' operator does not compare the content of the text present in the String objects. It only compares the references the 2 Strings are pointing to. The equals method is used when we need to compare the content of the text present in the String objects. This method returns true when two String objects hold the same content. The following section briefly discusses the different functions of the String class that can be used on the objects of String class.

2.4.7 Java String Functions

The following program explains the usage of the some of the basic String methods like ;

1. **compareTo**(String anotherString): Compares two strings lexicographically. It compares char values similar to the equals method. The compareTo method returns a negative integer if the first String object precedes the second string. It returns zero if the 2 strings being compared are equal. It returns a positive integer if the first String object follows the second string.
2. **charAt**(int index): Returns the character at the specified index.
3. **getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin): Copies characters from this string into the destination character array.
4. **length**(): Returns the length of this string.
5. **equals**(Object anObject): Compares this string to the specified object.

6. **equalsIgnoreCase**(String anotherString): Compares this String to another String, ignoring case considerations.
7. **toUpperCase**(): Converts all of the characters in this String to upper case using the rules of the default locale.
8. **toLowerCase**(): Converts all of the characters in this String to upper case using the rules of the default locale.
9. **concat**(String str): Concatenates the specified string to the end of this string.
10. **indexOf**(int ch): Returns the index within this string of the first occurrence of the specified character.
11. **indexOf**(int ch, int fromIndex): Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
12. **indexOf**(String str): Returns the index within this string of the first occurrence of the specified substring.
13. **indexOf**(String str, int fromIndex): Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
14. **lastIndexOf**(int ch): Returns the index within this string of the last occurrence of the specified character.
15. **lastIndexOf**(int ch, int fromIndex): Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
16. **lastIndexOf**(String str): Returns the index within this string of the rightmost occurrence of the specified substring.
17. **lastIndexOf**(String str, int fromIndex): Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
18. **substring**(int beginIndex): Returns a new string that is a substring of this string.
19. **substring**(int beginIndex, int endIndex): Returns a new string that is a substring of this string.
20. **replace**(char oldChar, char newChar): Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
21. **trim**(): Returns a copy of the string, with leading and trailing whitespace omitted.
22. **toString**(): This object (which is already a string!) is itself returned.

The following program demonstrates the use of String methods:

```
// Program to demonstrate String methods.
public class StringsDemo2
 {
   public static void main(String[] args)
```

```
{
        String str1 = "My name is bob";
        char str2[] = new char[str1.length()];
        String str3 = "bob";
        String str4 = "cob";
        String str5 = "BoB";
        String str6 = "bob";
        System.out.println("Length of the String str1 : " + str1.length());
        System.out.println("Character at position 3 is : "
                        + str1.charAt(3));
        str1.getChars(0, str1.length(), str2, 0);
        System.out.print("The String str2 is : ");
        for (int i = 0; i < str2.length; i++)
          {
            System.out.print(str2[i]);
          }
        System.out.println();
        System.out.print("Comparision Test : ");
        if (str3.compareTo(str4) < 0)
         {
            System.out.print(str3 + " < " + str4);
         }
        else if (str3.compareTo(str4) > 0)
         {
             System.out.print(str3 + " > " + str4);
         }
        else
         {
             System.out.print(str3 + " equals " + str4);
         }
        System.out.println();
        System.out.print("Equals Test");
        System.out.println("str3.equalsIgnoreCase(5) : "
                        + str3.equalsIgnoreCase(str5));
        System.out.println("str3.equals(6) : " + str3.equals(str6));
        System.out.println("str1.equals(3) : " + str1.equals(str3));
        str5.toUpperCase(); //Strings are immutable
        System.out.println("str5 : " + str5);
        String temp = str5.toUpperCase();
```

```
                System.out.println("str5 Uppercase: " + temp);
                temp = str1.toLowerCase();
                System.out.println("str1 Lowercase: " + str1);
                System.out.println("str1.concat(str4): " + str1.concat(str4));
                String str7temp = "  \t\n Now for some Search and Replace
Examples ";
                String str7 = str7temp.trim();
                System.out.println("str7 : " + str7);
                String newStr = str7.replace('s', 'T');
                System.out.println("newStr : " + newStr);
                System.out.println("indexof Operations on Strings");
                System.out.println("Index of p in " + str7 + " : " + str7.indexOf('p'));
                System.out.println("Index of for in " + str7 + " : " +
str7.indexOf("for"));
                System.out.println("str7.indexOf(for, 30) : " + str7.indexOf("for", 30));
                System.out.println("str7.indexOf('p', 30) : "+ str7.indexOf('p', 30));
                System.out.println("str7.lastIndexOf('p') : "+ str7.lastIndexOf('p'));
                System.out.println("str7.lastIndexOf('p', 4) : " + str7.lastIndexOf('p',
                4));
                System.out.print("SubString Operations on Strings");
                String str8 = "SubString Example";
                String sub5 = str8.substring(5); // "ring Example"
                String sub3_6 = str8.substring(3, 6); // "Str"
                System.out.println("str8 : " + str8);
                System.out.println("str8.substring(5) : " + sub5);
                System.out.println("str8.substring(3,6) : " + sub3_6);
        }
}
```

Output is:

```
Length of the String str1 : 14
Character at position 3 is : n
The String str2 is : My name is bob
Comparision Test : bob < cob
Equals Teststr3.equalsIgnoreCase(5) : true
str3.equals(6) : true
str1.equals(3) : false
str5 : BoB
str5 Uppercase: BOB
str1 Lowercase: My name is bob
```

str1.concat(str4): My name is bobcob
str7 : Now for some Search and Replace Examples
newStr : Now for Tome Search and Replace ExampleT
Indexof Operations on Strings
Index of p in Now for some Search and Replace Examples : 26
Index of for in Now for some Search and Replace Examples : 4
str7.indexOf(for, 30) : -1
str7.indexOf('p', 30) : 36
str7.lastIndexOf('p') : 36
str7.lastIndexOf('p', 4) : -1
SubString Operations on Stringsstr8 : SubString Example
str8.substring(5) : ring Example
str8.substring(3,6) : Str

2.4.8 Summary

The Java programming language allows you to define a class within another class. Such a class is called a *nested class.* Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are simply called *static nested classes*. Non-static nested classes are called *inner classes*. String is a special class built into the Java language defined in the java.lang package. The String class represents character strings. String literals in Java programs, such as "abc", are implemented as instances of this class. There are different functions of the String class that can be used on the objects of String class.

2.4.9 Review Questions

1. Differentiate between static nested and inner classes?
2. Can we access inner class objects outside the class it is defined in? How?
3. What is inheritance? Explain with the help of an example. How can we access base class constructors? Explain with example.

2.4.10 Suggested Readings

1. The Complete Reference by Herbert Scheild, Mc-Graw Hill
2. Programming with Java by E.Balagurusamy, Mc-Graw Hill
3. Java : A Beginner's Guide by Herbert Schildt, Mc-Graw Hill
4. Introduction to Java Programming by Y.Daniel Cians, Prentice Hall
5. Object Oriented Programming in Java by G.T. Thampi
6. Java Programming by C. Xavier

Web Resources

www.tutorialspoint.com/java/
www.javapoint.com/java-tutorial
www.w3schools.in/java-tutorial/
www.programiz.com/java-programming
www.learnjavaonline.org

Inheritance and Method Overriding

2.5.1 Introduction

Inheritance is the ability to define a new class that is a modified version of a previously-defined class. The primary advantage of this feature is that you can add new methods or instance variables to an existing class without modifying the existing class. In this chapter, we will see how to create a class that inherits the features of an existing class. The constructors in the base class are not inherited, but it is possible to call a base class constructor in a derived class. Finally a derived class can also define a method with same name and type signature as in the base class by using concept known as method overriding.

2.5.2 Objective

After reading this lesson you will be able to understand:

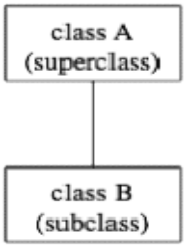
- Inheritance
- How to call base class constructors
- Method Overriding

2.5.3 Inheritance

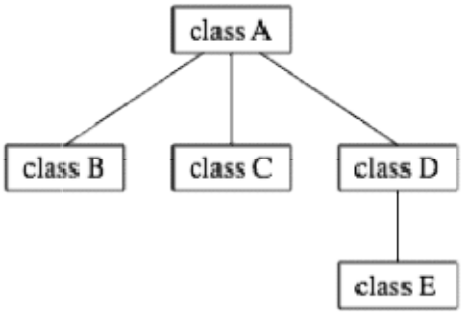
The language feature that is most often associated with object-oriented programming is **inheritance**. **Inheritance is the ability to define a new class that is a modified version of a previously-defined class (including built-in classes)**. The primary advantage of this feature is that you can add new methods or instance variables to an existing class without modifying the existing class. This is particularly useful for built-in classes, since you can't modify them even if you want to. The reason inheritance is called "inheritance" is that the new class inherits all the instance variables and methods of

the existing class. Extending this metaphor, the existing class is sometimes called the **parent** class. Inheritance means the parent class can have some child classes and the child classes can have or can inherit all the properties of parent class. The parent class can also be called as super class and the child class can be called as subclass. A keyword **extends** is used in order to make a child class of parent class. Following is an example of this :

```
Class A
{
//variables
}
class B extends A
{
// variables
}
```



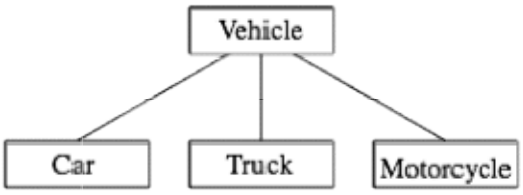
The term inheritance refers to the fact that one class can inherit part or all of its structure and behavior from another class. The class that does the inheriting is said to be a subclass of the class from which it inherits. If class B is a subclass of class A, we also say that class A is a superclass of class B. (Sometimes the terms derived class and base class are used instead of subclass and superclass; this is the common terminology in C++.) A subclass can add to the structure and behavior that it inherits. It can also replace or modify inherited behavior (though not inherited structure). The relationship between subclass and superclass is sometimes shown by a diagram in which the subclass is shown below, and connected to, its superclass.



Several classes can be declared as subclasses of the same superclass. The subclasses, which might be referred to as "sibling classes," share some structures and behaviors -- namely, the ones they inherit from their common superclass. The superclass expresses these shared structures and behaviors. In the diagram to the left, classes B, C, and D are sibling classes. Inheritance can also extend over several "generations" of classes. This is shown in the diagram, where class E is a subclass of class D which is itself a subclass of class A. In this case, class E is considered to be a subclass of class A, even though it is not a direct subclass. This whole set of classes forms a small class hierarchy.

Example: Vehicles

Let's look at an example. Suppose that a program has to deal with motor vehicles, including cars, trucks and motorcycles. (This might be a program used by a Department of Motor Vehicles to keep track of registrations.) The program could use a class named Vehicle to represent all types of vehicles. Since cars, trucks, and motorcycles are types of vehicles, they would be represented by subclasses of the Vehicle class, as shown in this class hierarchy diagram:



The Vehicle class would include instance variables such as registration Number and owner and instance methods such as transferOwnership(). These are variables and methods common to all vehicles. The three subclasses of Vehicle -- Car, Truck and Motorcycle -- could then be used to hold variables and methods specific to particular types of vehicles. The Car class might add an instance variable numberOfDoors, the Truck class might have numberOfAxles and the Motorcycle class could have a boolean variable hasSidecar. (Well, it could in theory at least, even if it might give a chuckle to the people at the Department of Motor Vehicles.) The declarations of these classes in a Java program would look, in outline, like this (although in practice, they would probably be public classes, defined in separate files):

```
class Vehicle
 {
   int registrationNumber;
   Person owner;  // (Assuming that a Person class has been defined!)
   void transferOwnership(Person newOwner)
    {
      . . .
```

```
    }
   . . .
 }
class Car extends Vehicle
 {
  int numberOfDoors;
  . . .
 }
class Truck extends Vehicle
 {
  int numberOfAxles;
  . . .
 }
class Motorcycle extends Vehicle
 {
  boolean hasSidecar;
  . . .
 }
```

Suppose that myCar is a variable of type Car that has been declared and initialized with the statement

Car myCar = new Car();

Given this declaration, a program could refer to myCar.numberOfDoors, since numberOfDoors is an instance variable in the class Car. But since class Car extends class Vehicle, a car also has all the structure and behavior of a vehicle. This means that myCar.registrationNumber, myCar.owner, and myCar.transferOwnership() also exist.

Now, in the real world, cars, trucks, and motorcycles are in fact vehicles. The same is true in a program. That is, an object of type Car or Truck or Motorcycle is automatically an object of type Vehicle too. This brings us to the following Important Fact:

"A variable that can hold a reference to an object of class A can also hold a reference to an object belonging to any subclass of A."

The practical effect of this in our example is that an object of type Car can be assigned to a variable of type Vehicle. That is, it would be legal to say

Vehicle myVehicle = myCar;

or even

Vehicle myVehicle = new Car();

After either of these statements, the variable myVehicle holds a reference to a Vehicle object that happens to be an instance of the subclass, Car. The object "remembers" that it is in fact a Car and not **just** a Vehicle.

You will be able to understand inheritance better after going through the following examples:

```
public class A
 {
   int i=0;
   void doSomething ()
    {
      i = 5;
    }
 }
class B extends A
 {
   int j = 0;
   void doSomethingMore ()
    {
     j = 10;
     i += j;
    }
 }
```

The class B above then has capabilities equivalent to class B1 shown below:

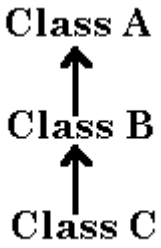
```
class B1
 {
   int i = 0;
   int j = 0;
   void doSomething ()
    {
      i = 5;
    }
   void doSomethingMore ()
    {
     j = 10;
     i += j;
    }
 }
```

An instance of either class B or B1 both possess i and j variables and the two methods. The class definition of class B is much smaller than B1 because the compiler

will link the members of the A base class to class B. Inheritance allows subclasses to build on a superclass to add new capabilities while the superclass is still available for situations where the new capabilities are not needed or applicable. Inheritance does more than reduce the size of the subclass definitions. We will see that the inheritance mechanism offers several new capabilities including the ability to re-define or override, a method in the superclass with a new one. We can now create instances of class **B** and access methods and data in both class **B** and class **A** (since they are *public* - access modifiers will be discussed later.)

```
...
B b = new B ();      //Create an instance of class B
b.doSomething ();    //Access class A methods
b.doSomethingMore ();//And class B methods
...
```

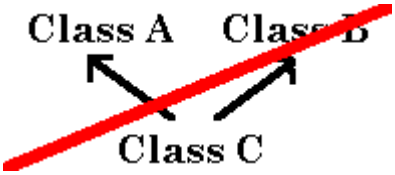
Another class can in turn inherit class **B**:




```
class C extends B
{
  int k;
  void doEvenMore ()
   {
    doSomething ();
    doSomethingMore ();
    k = i + j;
   }
}
```

Now an instance of class **C** can use the class C data and methods and also those of both classes **B** and **C**.

Note: Unlike C++, multiple inheritance is not allowed in Java:



Interfaces, discussed later, provide most of the benefits of multiple inheritance without the drawbacks.

Inheritance and Access

When inheritance is used to create a new (derived) class from an existing (base) class, everything in the base class is also in the derived class, it may not be accessible, however - the access in the derived class depends on the access in the base class:

base class access	**accessibility in derived class**
public	public
protected	protected
private	inaccessible
unspecified (package access)	unspecified (package access)

Note that private elements become inaccessible to the derived class - this does not mean that they disappear, or that that there is no way to affect their values, just that they can't be referenced by name in code within the derived class. Also note that a class can extend a class from a different package

2.5.4 Inheritance and Constructors - the super Keyword

Since a derived class object contains the elements of a base class object, it is reasonable to want to use the base class constructor as part of the process of constructing a derived class object. However it should be kept in mind that:

- constructors are "not inherited"
- in a sense, this is a moot point, since they would have a different name in the new class, and can't be called by name under any circumstances, so, for example, when one calls new Integer(int i) they shouldn't expect a constructor named Object(int i) to run

Within a derived class constructor, however, you can use super(*parameterList*) to call a base class constructor

- it must be done as the first line of a constructor
- therefore, you can't use both this() and super() in the same constructor function
- if you do not explicitly invoke a form of super-constructor, then super() (the form that takes no parameters) will run
- and for the superclass, its constructor will either explicitly or implicitly run a constructor for its superclass
- so, when an instance is created, *one constructor will run at every level of the inheritance chain*, all the way from Object up to the current class

```
class MyBase
 {
   private int x;
```

```
  public MyBase(int x)
   {
     this.x = x;
   }
  public int getX()
   {
     return x;
   }
  public void show()
   {
     System.out.println("x=" + x);
   }
}
class MyDerived extends MyBase
 {
   private int y;
   public MyDerived(int x)
    {
      super(x);
    }
   public MyDerived(int x, int y)
    {
      super(x);
      this.y = y;
    }
   public int getY()
    {
      return y;
    }
   public void show()
    {
     System.out.println("x = " + getX());
     System.out.println("y = " + y);
    }
 }
public class Inheritance1
 {
   public static void main(String[] args)
    {
```

```
    MyBase b = new MyBase(2);
    b.show();
    MyDerived d = new MyDerived(3, 4);
    d.show();
  }
}
```

Output is:

x=2
x = 3
y = 4

Code Explanation

- a MyDerived object has two constructors available, as well as both the getX and getY methods and the show method
- both MyDerived constructors call the super constructor to handle storage of x
- the show method in the derived class overrides the base class version
- x from the base class is not available in the derived class, since it is private in MyBase, so the show method in MyDerived must call getX() to obtain the value

2.5.5 Inheritance and Default Base Class Constructors

One base class constructor will *always* run when instantiating a new derived class object. if you do not explicitly call a base class constructor, the no-arguments base constructor will be automatically run, without the need to call it as super() but if you do explicitly call a base class constructor, the no-arguments base constructor will not be automatically run. The no-arguments (or no-args for short) constructor is often called the default constructor, since it the one that will run by default (and also because you are given it by default if you write no constructors). This will be clear from the following example:

```
class Purple
 {
  protected int i = 0;
  public Purple()
   {
    System.out.println("Purple() running and i = " + i);
   }
  public Purple(int i)
   {
    this.i = i;
    System.out.println("Purple(i) running and i = " + i);
   }
 }
```

```
class Violet extends Purple
 {
   Violet()
    {
      System.out.println("Violet(i) running and i = " + i);
    }
  Violet(int i)
   {
     super(i);
     System.out.println("Violet(i) running and i = " + i);
   }
 }
public class Inheritance2
 {
  public static void main(String[] args)
   {
     new Violet();
     new Violet(4);
   }
 }
```

Output is:

Purple() running and i = 0
Violet(i) running and i = 0
Purple() running and i = 4
Violet(i) running and i = 4

Each constructor prints a message so that we can follow the flow of execution. Note that using new Violet() causes Purple() to run and that new Violet(4) causes Purple(int i) to run. If your base class has constructors, but no no-arguments constructor, then the derived class must call one of the existing constructors with super(*args*), since there will be no default constructor in the base class. If the base class has a no-arguments constructor that is private, it will be there, but not be available, since private elements are hidden from the derived class. So, again, you must explicitly call an available form of base class constructor, rather than relying on the default. Try the above code with the Purple() constructor commented out or marked as private and see what happens.

2.5.6 Method Overriding

A subclass may want to provide a *new version* of a method in the superclass. In fact, this is usually the whole reason for creating a subclass. When a subclass method matches in name and in the number and type of arguments to the method in the

super-class (that is, the method *signatures* match), the subclass is said to *override* that method. In the code below, we see that subclass B overrides the method doSomething() in class A:

```
public class A
 {
   int i = 0;
   void doSomething (int k)
    {
      i = k;
    }
 }
class B extends A
 {
    int j = 0;
    void doSomething (ing k)
     {
       j = 10;
       i = 2 * k;
     }
 }
```

When we create an instance of class **B**, an invocation of the method doSomething() will result in a call to the doSomething() code in class **B** rather than **A**:

```
...
B b = new B ();   //Create an instance of class B
b.doSomething (); // Access class B method
                  // doSomething() will be
                  // called.
...
```

The real power of overriding, however, is illustrated by this code:

```
...
A ab = new B (); // Create an instance of class B
                 // but use A type reference.
ab.doSomething ();// Though the A type reference
                  // is used, the class B method
                  // doSomething() will be called.
...
```

Here we see that even though the superclass type variable **ab** references the subclass object, the subclass's method will be executed rather than the superclass's

overridden method. This is very useful when, for example, an array of the base class type contains references to various subclasses. Looping through the array and calling a method that is overriden will result in the method in the subclass being called rather than the method in the super-class. The following code illustrates this so-called **polymorphic** feature of object oriented languages:

```
...
  A [] a = new A[4]; // Class A type array
  a[0] =new B ();   // Create an instance of class B
                    // but use A type reference.
  a[1] =new B ();
  a[2] =new C ();   // Where class C is a subclass
                    // of class A or B and overrides doSomething().
  for (int i=0; i < 4; i++) {
    a[i].doSomething ();// Though the A type reference
                        // is used, the overriding
                        // doSomething () of the
                        // referenced object will be
                        // called.
  }
  ...
```

It is important to understand that even though the array is of the subclass A, the code used for the doSomething() methods will be that of the actual object that is referenced, not the code for the doSomething() method in the A base class .

2.5.7 Accessing Overridden method using super

As we saw before, you can create a method in the derived class with the same name as a base class method. The new method overrides (and hides) the original method. You can still call the base class method from within the derived class if necessary, by adding the super keyword and a dot in front of the method name. The base class version of the method is not available to outside code. You can view the super term as providing a reference to the base class object buried inside the derived class but you cannot do super.super. to back up two levels. You cannot change the return type when overriding a method, since this would make polymorphism impossible. The use of super to access the overridden method in the derived class will be clear from the following example:

```
class MyBase
 {
  private int x;
  public MyBase(int x)
   {
```

```
    this.x = x;
  }
  public int getX()
  {
    return x;
  }
  public void show()
  {
    System.out.println("x=" + x);
  }
}
class MyDerived extends MyBase
{
  int y;
  public MyDerived(int x)
  {
    super(x);
  }
  public MyDerived(int x, int y)
  {
    super(x);
    this.y = y;
  }
  public int getY()
  {
    return y;
  }
  public void show()
  {
    super.show();
    System.out.println("y = " + y);
  }
}
```

Output is:

```
x=2
x=3
y = 4
```

2.5.8 Summary

Inheritance is the ability to define a new class that is a modified version of a previously-defined class (including built-in classes). The primary advantage of this feature is that you can add new methods or instance variables to an existing class without modifying the existing class. Since a derived class object contains the elements of a base class object, it is reasonable to want to use the base class constructor as part of the process of constructing a derived class object. Within a derived class constructor, you can use super(*parameterList*) to call a base class constructor. One base class constructor will *always* run when instantiating a new derived class object. if you do not explicitly call a base class constructor, the no-arguments base constructor will be automatically run, without the need to call it as super() but if you do explicitly call a base class constructor, the no-arguments base constructor will not be automatically run. When a subclass method matches in name and in the number and type of arguments to the method in the super-class (that is, the method *signatures* match), the subclass is said to *override* that method.

2.5.9 Review Questions

1. What is inheritance? Explain with the help of an example. How can we access base class constructors? Explain with example.
2. What do you mean by method overriding?
3. How can you access overridden method in a derived class?

2.5.10 Suggested Readings

1. The Complete Reference by Herbert Scheild, Mc-Graw Hill
2. Programming with Java by E.Balagurusamy, Mc-Graw Hill
3. Java : A Beginner's Guide by Herbert Schildt, Mc-Graw Hill
4. Introduction to Java Programming by Y.Daniel Cians, Prentice Hall
5. Object Oriented Programming in Java by G.T. Thampi
6. Java Programming by C. Xavier

Web Resources

www.tutorialspoint.com/java/
www.javapoint.com/java-tutorial
www.w3schools.in/java-tutorial/
www.programiz.com/java-programming
www.learnjavaonline.org

Lesson No. 2.6 **Author : Kanwal Preet Singh**

Packages

2.6.1 Introduction

You've been using packages all along . Every time you use the import command, and every time you refer to a class by its full package name (java.awt.Color, for example), you've used packages. **A package is a way of grouping classes and interfaces.** The advantages of using a package are:

- Packages provide a way to hide classes, thus preventing other programs from accessing classes that are meant for internal use only.
- Two classes in two different packages can have the same name.
- The classes contained in the packages of other programs can easily be reused.

2.6.2 Objective

After reading this lesson you will be able to understand:

- What is a package
- Types of packages
- How to access classes stored in packages
- Creating your own packages

2.6.3 Types of Packages

Java packages are classified into two types:

- Java API packages and

- User Defined Packages

The Java API provides a large number of classes grouped together into different packages according to the functionality. Some of the commonly used Java API packages are:

- Java.lang Provides classes that are fundamental to the design of the Java programming language such as String, Math and basic runtime support for threads and processes.
- Java.util: Provides the collections framework, formatted printing and scanning, array manipulation utilities, event model, date and time facilities, internationalization and miscellaneous utility classes.
- Java.applet: Provides classes for creating and implementing applets.
- Java.awt: Provides set of classes for implementing a Graphical User Interface.

The programmers can also create their own packages to store their classes and interfaces.

2.6.4 Access class stored in a Package

Let's go over the specifics of how to use classes from other packages in your own programs. To use a class contained in a package, you can use one of three mechanisms:

- If the class you want to use is in the package java.lang (for example, System or Date), you can simply use the class name to refer to that class. The java.lang classes are automatically available to you in all your programs.
- If the class you want to use is in some other package, you can refer to that class by its full name, including any package names (for example, java.awt.Font).
- For classes that you use frequently from other packages, you can import individual classes or a whole package of classes. After a class or a package has been imported, you can refer to that class by its class name. What about your own classes in your own programs that don't belong to any package? The rule is that if you don't specifically define your classes to belong to a package, they're put into an unnamed default package. You can refer to those classes simply by class name from anywhere in your code.

2.6.4.1 Full Package and Class Names

To refer to a class in some other package, you can use its full name: the class name preceded by any package names. You do not have to import the class or the package to use it this way: **java.awt.Font f = new java.awt.Font()**

For classes that you use only once or twice in your program, using the full name makes the most sense. If, however, you use that class multiple times, or if the package name is really long with lots of subpackages, you'll want to import that class instead to save yourself some typing.

2.6.4.2 The import Command

To import classes from a package, use the import command, as you've used throughout the examples in this book. You can either import an individual class, like this:

import java.util.Vector;

or you can import an entire package of classes, using an asterisk (*) to replace the individual class names:

import java.awt.*

Actually, to be technically correct, this command doesn't import all the classes in a package-it only imports the classes that have been declared public, and even then only imports those classes that the code itself refers to. Note that the asterisk (*) in this example is not like the one you might use at a command prompt to specify the contents of a directory or to indicate multiple files. For example, if you ask to list the contents of the directory classes/java/awt/*, that list includes all the .class files and subdirectories, such as image and peer. Writing import java.awt.* imports all the public classes in that package, but does not import subpackages such as image and peer. To import all the classes in a complex package hierarchy, you must explicitly import each level of the hierarchy by hand. Also, you cannot indicate partial class names (for example, L* to import all the classes that begin with L). It's all the classes in a package or a single class. The import statements in your class definition go at the top of the file, before any class definitions (but after the package definition, as you'll see in the next section). So should you take the time to import classes individually or just import them as a group? It depends on how specific you want to be. Importing a group of classes does not slow down your program or make it any larger; only the classes you actually use in your code are loaded as they are needed. But importing a package does make it a little more confusing for readers of your code to figure out where your classes are coming from. Using individual imports or importing packages is mostly a question of your own coding style. Java's import command is not at all similar to the #include command in C-like languages, although they accomplish similar functions. The C preprocessor takes the contents of all the included files (and, in turn, the files they include, and so on) and stuffs them in at the spot where the #include was. The result is an enormous hunk of code that has far more lines than the original program did. Java's import behaves more like a linker; it tells the Java compiler and interpreter where (in which files) to find classes, variables, method names, and method definitions. It doesn't bring anything into the current Java program.

2.6.5 Name Conflicts

After you have imported a class or a package of classes, you can usually refer to a class name simply by its name, without the package identifier. I say "usually" because there's one case where you may have to be more explicit: when there are multiple classes with the same name from different packages.

Here's an example. Let's say you import the classes from two packages from two different programmers (Sachin and Rahul):

import Sachinsclasses.*;

import Rahulsclasses.*;

Inside Sachin's package is a class called Name. Unfortunately, inside Rahul's package there is also a class called Name that has an entirely different meaning and implementation. You would wonder whose version of Name would end up getting used if you referred to the Name class in your own program like this:

Name myName = new Name("Kanwal");

The answer is neither; the Java compiler will complain about a naming conflict and refuse to compile your program. In this case, despite the fact that you imported both classes, you still have to refer to the appropriate Name class by full package name:

Sachinsclasses.Name myName = new Sachinsclasses.Name("Kanwal");

2.6.6 CLASSPATH and Where Classes Are Located

Before I go on to explain how to create your own packages of classes, I'd like to make a note about how Java finds packages and classes when it's compiling and running your classes. For Java to be able to use a class, it has to be able to find it on the file system. Otherwise, you'll get an error that the class does not exist. Java uses two things to find classes: the package name itself and the directories listed in your CLASSPATH variable.

Package names map to directory names on the file system, so the class java.applet.Applet will actually be found in the applet directory, which in turn will be inside the java directory (java/applet/Applet.class, in other words).

Java looks for those directories, in turn, inside the directories listed in your CLASSPATH variable. When you installed the JDK, you had to set up a CLASSPATH variable to point to the various places where your Java classes live. CLASSPATH usually points to the java/lib directory in your JDK release, a class directory in your development environment if you have one, perhaps some browser-specific classes, and to the current directory. When Java looks for a class you've referenced in your source, it looks for the package and class name in each of those directories and returns an error if it can't find the class file. Most "cannot load class" errors result because of missed CLASSPATH variables.

Note: The Mac JDK doesn't use a CLASSPATH variable; it knows enough to be able to find the default classes and those contained in the current directory. However, if you do a lot of Java development, you may end up with classes and packages in other directories. The Java compiler contains a Preferences dialog box that lets you add directories to Java's search path.

2.6.7 Creating Your Own Packages

To create a package of classes, you have three basic steps to follow:

2.6.7.1 Picking a Package Name

The first step is to decide what the name of your package is going to be. The name you choose for your package depends on how you are going to be using those classes. Perhaps your package will be named after you, or perhaps after the part of the Java system you're working on (like graphics or hardware_interfaces). If you're intending your package to be

distributed to the Net at large, or as part of a commercial product, you'll want to use a package name (or set of package names) that uniquely identifies you or your organization or both.

One convention for naming packages that has been recommended by Sun is to use your Internet domain name with the elements reversed. So, for example, if Sun were following its own recommendation, its packages would be referred to using the name com.sun.java rather than just java. If your Internet domain name is pup.de.edu, your package name might be edu. pup.de (and you might add another package name onto the end of that to refer to the product or to you, specifically). The idea is to make sure your package name is unique. Although packages can hide conflicting class names, the protection stops there. There's no way to make sure your package won't conflict with someone else's package if you both use the same package name.

By convention, package names tend to begin with a lowercase letter to distinguish them from class names. Thus, for example, in the full name of the built-in String class, java.lang.String, it's easier to separate the package name from the class name visually. This convention helps reduce name conflicts.

2.6.7.2 Create the Directory Structure

Step two in creating packages is to create a directory structure on your disk that matches the package name. If your package has just one name (mypackage), you'll only have to create a directory for that one name. If the package name has several parts, however, you'll have to create directories within directories. For the package name edu.pup.de, you'll need to create an edu directory and then create a pup directory inside edu and a de directory inside pup. Your classes and source files can then go inside the pup directory.

2.6.7.3 Use package to Add Your Class to a Package

The final step to putting your class inside packages is to add the package command to your source files. The package command says "this class goes inside this package," and is used like this:

package package_name;

for example:

package myclasses;

The single package command, if any, must be the first line of code in your source file, after any comments or blank lines and before any import commands. As mentioned before, if your class doesn't have a package command in it, that class is contained in the default package and can be used by any other class. But once you start using packages, you should make sure all your classes belong to some package to reduce the chance of confusion about where your classes belong.

2.6.8 Packages and Class Protection

When referring to classes and their relationship to other classes in other packages, you only have two Ps to worry about: package and public. By default, classes have package

protection, which means that the class is available to all the other classes in the same package but is not visible or available outside that package-not even to subpackages. It cannot be imported or referred to by name; classes with package protection are hidden inside the package in which they are contained. Package protection comes about when you define a class as you have throughout this book, like this:

```
class TheHiddenClass extends AnotherHiddenClass
 {
  ...
 }
```

To allow a class to be visible and importable outside your package, you'll want to give it public protection by adding the public modifier to its definition:

```
public class TheVisibleClass
 {
  ...
 }
```

Classes declared as public can be imported by other classes outside the package. Note that when you use an import statement with an asterisk, you import only the public classes inside that package. Hidden classes remain hidden and can be used only by the other classes in that package.

Why would you want to hide a class inside a package? For the same reason you want to hide variables and methods inside a class: so you can have utility classes and behavior that are useful only to your implementation, or so you can limit the interface of your program to minimize the effect of larger changes. As you design your classes, you'll want to take the whole package into consideration and decide which classes will be declared public and which will be hidden.

The following example shows two classes that illustrate this point. The first is a public class that implements a linked list; the second is a private node of that list.

```
//The public class LinkedList.
 package  collections;
  public class  LinkedList
       {
         private Node  root;
         public  void  add(Object o)
          {
           root = new Node(o, root);
          }
              . . .
        }
  class  Node
```

```
{  // not public
 private Object  contents;
 private Node    next;
 Node(Object o, Node n)
  {
   contents = o;
   next     = n;
  }
  . . .
}
```

Note: Notice here that I'm including two class definitions in one file. You can include as many class definitions per file as you want, but only one of them can be declared public, and that filename must have the same name as the one public class. When Java compiles the file, it'll create separate .class files for each class definition inside the file. In reality, I find the one-to-one correspondence of class definition to file much more easily maintained because I don't have to go searching around for the definition of a class.

The public LinkedList class provides a set of useful public methods (such as add()) to any other classes that might want to use them. These other classes don't need to know about any support classes LinkedList needs to get its job done. Node, which is one of those support classes, is therefore declared without a public modifier and will not appear as part of the public interface to the collections package.

Just because Node isn't public doesn't mean LinkedList won't have access to it once it's been imported into some other class. Think of protections not as hiding classes entirely, but more as checking the permissions of a given class to use other classes, variables, and methods. When you import and use LinkedList, the Node class will also be loaded into the system, but only instances of LinkedList will have permission to use it.

One of the great powers of hidden classes is that even if you use them to introduce a great deal of complexity into the implementation of some public class, all the complexity is hidden when that class is imported or used. Thus, creating a good package consists of defining a small, clean set of public classes and methods for other classes to use, and then implementing them by using any number of hidden (package) support classes.

2.6.9 Summary

A package is a way of grouping classes and interfaces. Java packages are classified into two types: Java API packages and User Defined Packages. The Java API provides a large number of classes grouped together into different packages according to the functionality. The programmers can also create their own packages to store their classes and interfaces. To use a class contained in a package, you can use the fully classified class name or you can import the package containing that class. After you have imported a class or a package of classes, you can usually refer to a class name simply by its name, without the

package identifier. For Java to be able to use a class, it has to be able to find it on the file system. Otherwise, you'll get an error that the class does not exist. Java uses two things to find classes: the package name itself and the directories listed in your CLASSPATH variable. You can also create your own packages. To do that, first think of a suitable package name, create the directory structure, and then write the package_name at the top of your file to include the classes in a particular package.

2.6.10 Review Questions

1. What is a package? What are its advantages?
2. What are the different categories of packages
3. How can you create your own package?
4. How can you use a class stored in some other package?

2.6.11 Suggested Readings

1. The Complete Reference by Herbert Scheild, Mc-Graw Hill
2. Programming with Java by E.Balagurusamy, Mc-Graw Hill
3. Java : A Beginner's Guide by Herbert Schildt, Mc-Graw Hill
4. Introduction to Java Programming by Y.Daniel Cians, Prentice Hall
5. Object Oriented Programming in Java by G.T. Thampi
6. Java Programming by C. Xavier

Web Resources

www.tutorialspoint.com/java/
www.javapoint.com/java-tutorial
www.w3schools.in/java-tutorial/
www.programiz.com/java-programming
www.learnjavaonline.org

Java Abstract class and Interface

2.7.1 Introduction

Java is an Object Oriented Language. There are loads of built in classes stored in packages and if we want to use the built in features of Java, we have to inherit those built in classes. But in Java, only single inheritance is allowed, that is, a class can inherit only one other class. Multiple inheritance is not allowed in Java. To overcome this problem, we make use of interfaces. In this lesson, we will discuss the concepts of abstract class and interface. Abstract class is a class which contains one or more abstract methods, which has to be implemented by sub classes. An abstract class can contain non abstract methods also i.e. abstract class may contain concrete methods. A Java Interface can contain only method declarations and public static final constants and doesn't contain their implementation. The classes which implement the Interface must provide the method definition for all the methods present.

2.7.2 Objective

After reading this lesson you will be able to understand:

- Abstract classes
- Interfaces

2.7.3 Abstract Class

Java Abstract classes are used to declare common characteristics of subclasses. An abstract class cannot be instantiated. It can only be used as a superclass for other classes that extend the abstract class. Abstract classes are declared with the abstract keyword. Abstract classes are used to provide a template or design for concrete subclasses down the inheritance tree.Like any other class, an abstract class can contain fields that describe the characteristics and methods that describe the actions that a class can perform. An abstract class can include methods that contain no implementation. These are called abstract methods. The abstract method declaration

must then end with a semicolon rather than a block. If a class has any abstract methods, whether declared or inherited, the entire class must be declared abstract. Abstract methods are used to provide a template for the classes that inherit the abstract methods.

Abstract classes cannot be instantiated; they must be subclassed, and actual implementations must be provided for the abstract methods. Any implementation specified can, of course, be overridden by additional subclasses. An object must have an implementation for all of its methods. You need to create a subclass that provides an implementation for the abstract method.

An abstract class - Vehicle might be specified as abstract to represent the general abstraction of a vehicle, as creating instances of the class would not be meaningful.

```
abstract class Vehicle
 {
        int numofGears;
        String color;
        abstract boolean hasDiskBrake();
        abstract int getNoofGears();
}
```

Example of a shape class as an abstract class

```
abstract class Shape
 {
        public String color;
        public Shape()
          {
          }
        public void setColor(String c)
          {
                color = c;
          }
        public String getColor()
          {
                return color;
          }
        abstract public double area();
  }
```

We can also implement the generic shapes class as an abstract class so that we can draw lines, circles, triangles etc. All shapes have some common fields and methods, but each can, of course, add more fields and methods. The abstract class guarantees that each shape will have the same set of basic properties. We declare this

class abstract because there is no such thing as a generic shape. There can only be concrete shapes such as squares, circles, triangles etc.

```
public class Point extends Shape
 {
        static int x, y;
        public Point()
         {
                x = 0;
                y = 0;
         }
        public double area()
         {
                return 0;
         }
        public double perimeter()
         {
                return 0;
         }
        public static void print()
         {
                System.out.println("point: " + x + "," + y);
         }
        public static void main(String args[])
          {
                Point p = new Point();
                p.print();
          }
   }
```

Output

point: 0, 0

Notice that, in order to create a Point object, its class cannot be abstract. This means that all of the abstract methods of the Shape class must be implemented by the Point class. The subclass must define an implementation for every abstract method of the abstract superclass or the subclass itself will also be abstract. Similarly other shape objects can be created using the generic Shape Abstract class.A big Disadvantage of using abstract classes is not able to use multiple inheritance. In the sense, when a class extends an abstract class, it can't extend any other class.

2.7.4 Java Interface

In Java, this multiple inheritance problem is solved with a powerful construct called interfaces. Interface can be used to define a generic template and then one or

more abstract classes to define partial implementations of the interface. **Interfaces just specify the method declaration (implicitly public and abstract) and can only contain fields (which are implicitly public static final).** Interface definition begins with a keyword interface. An interface like that of an abstract class cannot be instantiated. To create an interface definition:

- define it like a Java class, in its own file that matches the interface name
- use the keyword interface instead of class
- declare methods using the same approach as abstract methods
    - note the semicolon after each method declaration - and that no executable code is supplied(and no curly braces)
    - the elements will automatically be public and abstract and cannot have any other state; it is OK to specify those terms, but not necessary (usually public is specified and abstract is not - that makes it easy to copy the list of methods, paste them into a class, and modify them)
- The access level for the entire interface is usually public
    - it may be omitted, in which case the interface is only available to other classes in the same package (i.e., in the same directory)
    - note, for the sake of completeness, there are situations where the interface definition could be protected or private; these involve what are called *inner classes*

Syntax

```
[modifiers] interface InterfaceName
{
  // declaring methods
  [public abstract] returnType methodName1(arguments);
  // defining constants
  [public static final]
  type propertyName = value;
}
```

Example:

```
public interface Printable
{
  void printAll();
}
```

This interface requires only one method. Any class implementing Printable must contain a public void printall() method in order to compile Because the above interface is defined as public, its definition must be in its own file, even though that file will be tiny An interface definition may also define properties that are automatically public static final - these are used as constants.

2.7.5 Implementing Interfaces

A class definition may, in addition to whatever else it does, implement one or more interfaces. Once a class states that it implements an interface, it must supply all the methods defined for that interface, complete with executable code

Note: it actually does not have to implement all of them, but in that case the class cannot be instantiated- it must be declared as an abstract class that can only be used as a base class (where some derived class would then fully implement the interface)

To implement an interface:

- add that the class implements the interface to the class declaration
- add the methods specified by the interface to the body of the class
    - note that you *do* need to specify the access terms on methods in a class that implements an interface

Syntax

```
[modifiers] class ClassName implements InterfaceName
{
  any desired properties
  // implement required methods
  [modifiers] returnType methodName1(arguments)
   {
     executable code
   }
  any other desired methods
}
```

It is important to note that a class may implement an interface in addition to whatever else it might do, so it could have additional properties and methods not associated with the interface A class may implement more than one interface - that merely adds to the list of required methods

- use a comma-separated list for the interface names

Syntax

```
[modifiers] class ClassName implements Interface1Name, Interface2Name
{
  // must implement all methods from all implemented interfaces
 }
```

Multiple Inheritance is allowed when extending interfaces i.e. one interface can extend none, one or more interfaces. Java does not support multiple inheritance, but it allows you to extend one class and implement many interfaces.If a class that implements an interface does not define all the methods of the interface, then it must be declared abstract and the method definitions must be provided by the subclass that extends the abstract class.

Below is an example of a Shape interface

```
interface Shape
{
        public double area();
        public double volume();
}
```

Below is a Point class that implements the Shape interface.

```
public class Point implements Shape
{
        static int x, y;
        public Point()
        {
                x = 0;
                y = 0;
        }
        public double area()
        {
                return 0;
        }
        public double volume()
        {
                return 0;
        }
        public static void print()
        {
                System.out.println("point: " + x + "," + y);
        }
        public static void main(String args[])
        {
                Point p = new Point();
                p.print();
        }
 }
```

Similarly, other shape objects can be created by interface programming by implementing generic Shape Interface. Below is a java interfaces program showing the power of interface programming in java

Note: The method toString in class A1 is an overridden version of the method defined in the class named Object. The classes B1 and C1 satisfy the interface contract. But since the class D1 does not define all the methods of the implemented interface I2, the class D1 is declared abstract. Also, i1.methodI2() produces a compilation error as the method is not declared in I1 or any of its super interfaces if present. Hence a

downcast of interface reference I1 solves the problem as shown in the program. The same problem applies to i1.methodA1(), which is again resolved by a downcast.

When we invoke the toString() method which is a method of an Object, there does not seem to be any problem as every interface or class extends Object and any class can override the default toString() to suit your application needs. ((C1)o1).methodI1() compiles successfully, but produces a ClassCastException at runtime. This is because B1 does not have any relationship with C1 except they are "siblings". You can't cast siblings into one another.

When a given interface method is invoked on a given reference, the behavior that results will be appropriate to the class from which that particular object was instantiated. This is runtime polymorphism based on interfaces and overridden methods.

```
interface I1
{
        void methodI1(); // public static by default
}
interface I2 extends I1
{
        void methodI2(); // public static by default
}
class A1
{
        public String methodA1()
        {
                String strA1 = "I am in methodC1 of class A1";
                return strA1;
        }
        public String toString()
        {
                return "toString() method of class A1";
        }
}
class B1 extends A1 implements I2
{
        public void methodI1()
        {
                System.out.println("I am in methodI1 of class B1");
        }
        public void methodI2()
        {
```

```
                System.out.println("I am in methodI2 of class B1");
        }
 }
class C1 implements I2
{
        public void methodI1()
        {
                System.out.println("I am in methodI1 of class C1");
        }
        public void methodI2()
        {
                System.out.println("I am in methodI2 of class C1");
        }
}
// Note that the class is declared as abstract as it does not
// satisfy the interface contract
abstract class D1 implements I2
{
        public void methodI1()
        {
        }
        // This class does not implement methodI2() hence declared abstract.
}
public class InterFaceEx
{
        public static void main(String[] args)
        {
        I1 i1 = new B1();
        i1.methodI1(); // OK as methodI1 is present in B1
        // i1.methodI2(); Compilation error as methodI2 not present in I1
        // Casting to convert the type of the reference from type I1 to type I2
                ((I2) i1).methodI2();
                I2 i2 = new B1();
                i2.methodI1(); // OK
                i2.methodI2(); // OK
                // Does not Compile as methodA1() not present in interface
               reference I1
                // String var = i1.methodA1();
                // Hence I1 requires a cast to invoke methodA1
                String var2 = ((A1) i1).methodA1();
```

```
            System.out.println("var2 : " + var2);
            String var3 = ((B1) i1).methodA1();
            System.out.println("var3 : " + var3);
            String var4 = i1.toString();
            System.out.println("var4 : " + var4);
            String var5 = i2.toString();
            System.out.println("var5 : " + var5);
            I1 i3 = new C1();
            String var6 = i3.toString();
            System.out.println("var6 : " + var6); //It prints the Object toString()
            //method
            Object o1 = new B1();
            // o1.methodI1(); does not compile as Object class does not define
            // methodI1()
            // To solve the probelm we need to downcast o1 reference. We can
            do it
            // in the following 4 ways
            ((I1) o1).methodI1(); // 1
            ((I2) o1).methodI1(); // 2
            ((B1) o1).methodI1(); // 3
            /*
     * B1 does not have any relationship with C1 except they are "siblings".
              * Well, you can't cast siblings into one another.
            */
            // ((C1)o1).methodI1(); Produces a ClassCastException
        }
 }
```

Output

```
I am in methodI1 of class B1
I am in methodI2 of class B1
I am in methodI1 of class B1
I am in methodI2 of class B1
var2 : I am in methodC1 of class A1
var3 : I am in methodC1 of class A1
var4 : toString() method of class A1
var5 : toString() method of class A1
var6 : C1@190d11
I am in methodI1 of class B1
I am in methodI1 of class B1
I am in methodI1 of class B1
```

2.7.6 Interfaces and Inheritance

If a class implements an interface, then all subclasses of it will also automatically implement the interface. They are guaranteed to have the necessary methods available. It is a good practice to specify that the derived class implements the interface, just for self-documentation of the code (also for purposes of javadoc, if the base class is not in the same group of files)

An interface definition can inherit from another interface

- the new interface then adds properties and methods to the existing (base) definition
- a class that implements the new interface must implement all methods from the base interface as well as all the additional methods from the new interface definition

2.7.7 Summary

Abstract class is a class which contains one or more abstract methods, which has to be implemented by sub classes. An abstract class can contain no abstract methods also i.e. abstract class may contain concrete methods. A Java Interface can contain only method declarations and public static final constants and doesn't contain their implementation. The classes which implement the Interface must provide the method definition for all the methods present. Abstract class definition begins with the keyword "abstract" keyword followed by Class definition. An Interface definition begins with the keyword "interface". Abstract classes are useful in a situation when some general methods should be implemented and specialization behavior should be implemented by subclasses. Interfaces are useful in a situation when all its properties need to be implemented by subclasses. All variables in an Interface are by default - public static final while an abstract class can have instance variables. An interface is also used in situations when a class needs to extend an other class apart from the abstract class. In such situations its not possible to have multiple inheritance of classes. An interface on the other hand can be used when it is required to implement one or more interfaces. Abstract class does not support Multiple Inheritance whereas an Interface supports multiple Inheritance. An Interface can only have public members whereas an abstract class can contain private as well as protected members. A class implementing an interface must implement all of the methods defined in the interface, while a class extending an abstract class need not implement any of the methods defined in the abstract class. The problem with an interface is, if you want to add a new feature (method) in its contract, then you MUST implement those methods in all of the classes which implement that interface. However, in the case of an abstract class, the method can be simply implemented in the abstract class and the same can be called by its subclass. Neither Abstract classes nor Interface can be instantiated.

2.7.8 Review Questions

1. What is an abstract class? What is the purpose of defining an abstract class?
2. Is multiple inheritance possible in Java? How do we overcome this problem?
3. What is an interface? How can you define and implement an interface?

2.7.9 Suggested Readings

1. The Complete Reference by Herbert Scheild, Mc-Graw Hill
2. Programming with Java by E.Balagurusamy, Mc-Graw Hill
3. Java : A Beginner's Guide by Herbert Schildt, Mc-Graw Hill
4. Introduction to Java Programming by Y.Daniel Cians, Prentice Hall
5. Object Oriented Programming in Java by G.T. Thampi
6. Java Programming by C. Xavier

Web Resources

www.tutorialspoint.com/java/
www.javapoint.com/java-tutorial
www.w3schools.in/java-tutorial/
www.programiz.com/java-programming
www.learnjavaonline.org

Exception Handling

2.8.1 Introduction

Exception handling is a mechanism that allows Java programs to handle various exceptional conditions, such as semantic violations of the language and program-defined errors, in a robust way. When an exceptional condition occurs, an *exception* is thrown. If the Java virtual machine or run-time environment detects a semantic violation, the virtual machine or run-time environment implicitly throws an exception. Alternately, a program can throw an exception explicitly using the throw statement. After an exception is thrown, control is transferred from the current point of execution to an appropriate catch clause of an enclosing try statement. The catch clause is called an exception handler because it handles the exception by taking whatever actions are necessary to recover from it.

2.8.2 Objective

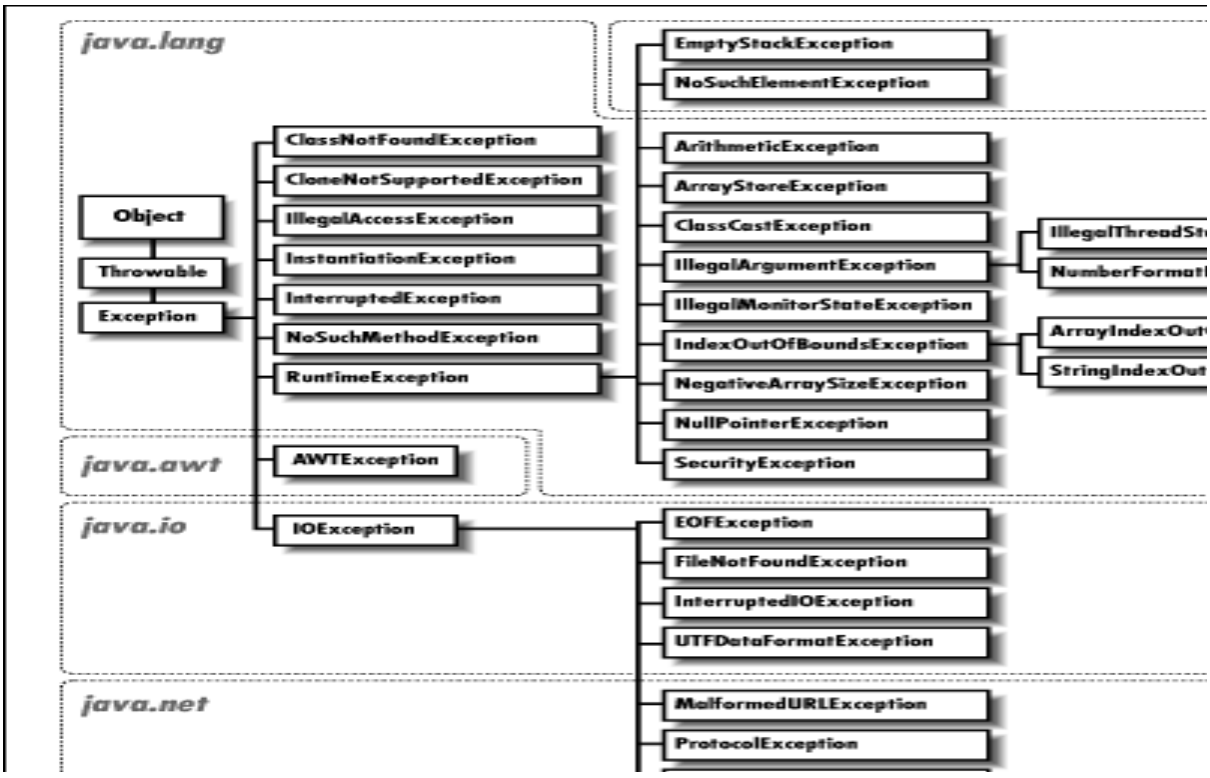
After reading this lesson you will be able to understand:

- What are Exceptions?
- How to handle Exceptions
- Different types of Exceptions
- How to create your own Exceptions
- throw and throws clause

2.8.3 Exceptions and Error Classes

Exceptions are represented by instances of the class java.lang.Exception and its subclasses. Subclasses of Exception can hold specialized information (and possibly behavior) for different kinds of exceptional conditions. However, more often they are simply "logical" subclasses that exist only to serve as a new exception type (more on that later). The following figure shows the subclasses of Exception; these classes are defined in various packages in the Java API, as indicated in the diagram.

Figure : Java exception classes



An Exception object is created by the code at the point where the error condition arises. It can hold whatever information is necessary to describe the exceptional condition, including a full stack trace for debugging. The exception object is passed, along with the flow of control, to the handling block of code. This is where the terms "throw" and "catch" come from: the Exception object is thrown from one point in the code and caught by the other, where execution resumes.

The Java API also defines the java.lang.Error class for unrecoverable errors. You needn't worry about these errors (i.e., you do not have to catch them); they normally indicate linkage problems or virtual machine errors. An error of this kind usually causes the Java interpreter to display a message and exit.

2.8.4 Exception Handling

The try/catch guarding statements wrap a block of code and catch designated types of exceptions that occur within it:

```
try
 {
   readFromFile("student");
   ...
 }
catch ( Exception e )
 {
   // Handle error
   System.out.println( "Exception while reading file: " + e );
   ...
 }
```

In the above example, exceptions that occur within the body of the try statement are directed to the catch clause for possible handling. The catch clause acts like a method; it specifies an argument of the type of exception it wants to handle and if it's invoked, the Exception object is passed into its body as an argument. Here we receive the object in the variable e and print it along with a message.

A try statement can have multiple catch clauses that specify different specific types (subclasses) of Exception:

```
try
 {
   readFromFile("student");
   ...
 }
catch ( FileNotFoundException e )
 {
   // Handle file not found
   ...
 }
catch ( IOException e )
 {
   // Handle read error
   ...
 }
catch ( Exception e )
 {
   // Handle all other errors
```

```
   ...
}
```

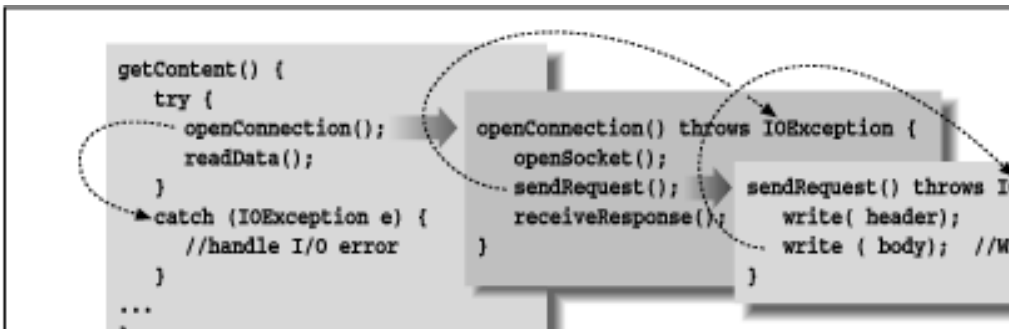
The catch clauses are evaluated in order, and the first possible (assignable) match is taken. At most one catch clause is executed, which means that the exceptions should be listed from most specific to least. In the above example, we'll assume that the hypothetical readFromFile() can throw two different kinds of exceptions: one that indicates the file is not found; the other indicates a more general read error. Any subclass of Exception is assignable to the parent type Exception, so the third catch clause acts like the default clause in a switch statement and handles any remaining possibilities.

It should be obvious, but one beauty of the try/catch statement is that any statement in the try block can assume that all previous statements in the block succeeded. A problem won't arise suddenly because a programmer forgot to check the return value from some method. If an earlier statement fails, execution jumps immediately to the catch clause; later statements are never executed.

2.8.5 Exception propagation

What if we hadn't caught the exception? Where would it have gone? Well, if there is no enclosing try/catch statement, the exception pops to the top of the method in which it appeared and is, in turn, thrown from that method. In this way, the exception bubbles up until it's caught, or until it pops out of the top of the program, terminating it with a run-time error message. Let's look at another example. In the following figure the method getContent() invokes the method openConnection() from within a try/catch statement. openConnection(), in turn, invokes the method sendRequest(), which calls the method write() to send some data.

Figure: Exception propagation



In this figure, the second call to write() throws an IOException. Since sendRequest() doesn't contain a try/catch statement to handle the exception, it's thrown again, from the point that it was called in the method openConnection(). Since openConnection() doesn't catch the exception either, it's thrown once more. Finally it's caught by the try statement in getContent() and handled by its catch clause.

Since an exception can bubble up quite a distance before it is caught and handled, we may need a way to determine exactly where it was thrown. All exceptions

can dump a *stack trace* that lists their method of origin and all of the nested method calls that it took to arrive there, using the printStackTrace() method.

```
try
 {
   // complex task
 }
 catch ( Exception e )
  {
   // dump information about exactly where the exception ocurred
   e.printStackTrack( System.err );
   ...
 }
```

2.8.6 The *throws* Clause and *checked* Exceptions

I mentioned earlier that Java makes us be explicit about our error handling. But Java is programmer-friendly and it's not possible to require that every conceivable type of error be handled in every situation. So, Java exceptions are divided into two categories: *checked exceptions* and *unchecked exceptions*. Most application level exceptions are checked, which means that any method that throws one, either by generating it itself (as we'll discuss below) or by passively ignoring one that occurs within it, must declare that it can throw that type of exception in a special throws clause in its method declaration. For now all you need know is that methods have to declare the checked exceptions they can throw or allow to be thrown. In the figure above, notice that the methods openConnection() and sendRequest() both specify that they can throw an IOException. If we had to throw multiple types of exceptions we could declare them separated with commas:

```
void readFile( String s ) throws IOException, InterruptedException
 {
   ...
 }
```

The throws clause tells the compiler that a method is a possible source of that type of checked exception and that anyone calling that method must be prepared to deal with it. The caller may use a *try/catch* block to catch it, or it may, itself, declare that it can throw the exception.

Exceptions that are subclasses of the java.lang.RuntimeException class are unchecked. See the first figure for the subclasses of RuntimeException. It's not a compile-time error to ignore the possibility of these exceptions being thrown; additionally, methods don't have to declare they can throw them. In all other respects, run-time exceptions behave the same as other exceptions. We are perfectly free to catch them if we wish; we simply aren't required to.

Checked exceptions: Exceptions a reasonable application should try to handle gracefully.
Unchecked exception (Runtime exceptions): Exceptions from which we would not normally expect our software to try to recover.
The category of checked exceptions includes application-level problems like missing files and unavailable hosts. As good programmers, we should design software to recover gracefully from these kinds of conditions. The category of unchecked exceptions includes problems such as "out of memory" and "array index out of bounds." While these may indicate application-level programming errors, they can occur almost anywhere and aren't generally easy to recover from. Fortunately, because there are unchecked exceptions, you don't have to wrap every one of your array-index operations in a try/catch statement.

2.8.7 Throwing Exceptions

We can throw our own exceptions: either instances of Exception or one of its predefined subclasses, or our own specialized subclasses. All we have to do is create an instance of the Exception and throw it with the throw statement:

```
throw new Exception();
```

Execution stops and is transferred to the nearest enclosing try/catch statement. (Note that there is little point in keeping a reference to the Exception object we've created here.) An alternative constructor of the Exception class lets us specify a string with an error message:

```
throw new Exception("Something really bad happened");
```

By convention, all types of Exception have a String constructor like this. Note that the String message above is somewhat facetious and vague. Normally you won't be throwing a plain old Exception, but a more specific subclass. For example:

```
public void checkRead( String s )
 {
   if ( new File(s).isAbsolute() || (s.indexOf("..") != -1) )
       throw new SecurityException("Access to file : "+ s +" denied.");
 }
```

In the above, we partially implement a method to check for an illegal path. If we find one, we throw a SecurityException, with some information about the transgression.

2.8.8 Creating your own Exceptions

You can also create your own exception classes by simply extending the Exception class. Of course, we could include whatever other information is useful in our own specialized subclasses of Exception. Often though, just having a new type of exception is good enough, because it's sufficient to help direct the flow of control. For example, if we are building a parser, we might want to make our own kind of exception to indicate a particular kind of failure.

```
class ParseException extends Exception
{
   ParseException()
    {
      super();
    }
   ParseException( String desc )
    {
      super( desc );
    }
 }
```

The body of our exception class here simply allows a ParseException to be created in the conventional ways that we have created exceptions above. Now that we have our new exception type, we we might guard for it in the following kind of situation:

```
// Somewhere in our code
...
try
 {
   parseStream( input );
 }
catch ( ParseException pe )
 {
   // Bad input...
 }
catch ( IOException ioe )
 {
   // Low level communications problem
 }
```

As you can see, although our new exception doesn't currently hold any specialized information about the problem (it certainly could), it does let us distinguish a parse error from an arbitrary communications error in the same chunk of code. You might call this kind of specialization of an exception to be making a "logical" exception.

2.8.9 Re-throwing exceptions

After an exception is caught, it can be rethrown if is appropriate. The one choice that you have to make when rethrowing an exception concerns the location from where the stack trace says the object was thrown. You can make the rethrown exception appear to have been thrown from the location of the original exception throw, or from the location of the current rethrow. To rethrow an exception and have the stack trace indicate the original location, all you have to do is rethrow the exception:

```
try
{
   cap(0);
}
catch(ArithmeticException e)
{
   throw e;
}
```

To arrange for the stack trace to show the actual location from which the exception is being rethrown, you have to call the exception's fillInStackTrace() method. This method sets the stack trace information in the exception based on the current execution context. Here's an example using the fillIn-StackTrace() method:

```
try
{
   cap(0);
}
catch(ArithmeticException e)
{
   throw (ArithmeticException)e.fillInStackTrace();
}
```

It is important to call fillInStackTrace() on the same line as the throw statement, so that the line number specified in the stack trace matches the line on which the throw statement appears. The fillInStackTrace() method returns a reference to the Throwable class, so you need to cast the reference to the actual type of the exception.

2.8.10 try Creep

The try statement imposes a condition on the statements they guard. It says that if an exception occurs within it, the remaining statements will be abandoned. This has consequences for local variable initialization. If the compiler can't determine whether a local variable assignment we placed inside a try/catch block will happen, it won't let us use the variable:

```
void myMethod()
{
   int student;

   try
    {
      student = getResults();
    }
```

```
   catch ( Exception e )
    {
       ...
    }
    int bar = student;  // Compile time error--student may not
                        // have been initialized
```

In the above example, we can't use student in the indicated place because there's a chance it was never assigned a value. One obvious option is to move the assignment inside the try statement:

```
try
 {
   student = getResults();
   int bar = student;  // Okay because we only get here
                       // if previous assignment succeeds
 }
catch ( Exception e )
 {
   ...
 }
```

Sometimes this works just fine. However, now we have the same problem if we want to use bar later in myMethod(). If we're not careful, we might end up pulling everything into the try statement. The situation changes if we transfer control out of the method in the catch clause:

```
try
 {
   student = getResults();
 }
catch ( Exception e )
 {
   ...
   return;
 }

int bar = student;  // Okay because we only get here
                    // if previous assignment succeeds
```

Your code will dictate its own needs; you should just be aware of the options.

2.8.11 The finally Clause

What if we have some clean up to do before we exit our method from one of the catch clauses? To avoid duplicating the code in each catch branch and to make the

cleanup more explicit, Java supplies the finally clause. A finally clause can be added after a try and any associated catch clauses. Any statements in the body of the finally clause are guaranteed to be executed, no matter why control leaves the try body:

```
try
 {
   // Do something here
 }
catch ( FileNotFoundException e )
 {
   ...
 }
catch ( IOException e )
 {
   ...
 }
catch ( Exception e )
 {
   ...
 }
finally
 {
   // Cleanup here
 }
```

In the above example the statements at the cleanup point will be executed eventually, no matter how control leaves the try. If control transfers to one of the catch clauses, the statements in finally are executed after the catch completes. If none of the catch clauses handles the exception, the finally statements are executed before the exception propagates to the next level.

If the statements in the try execute cleanly, or even if we perform a return, break, or continue, the statements in the finally clause are executed. To perform cleanup operations, we can even use try and finally without any catch clauses:

```
try
 {
   // Do something here
   return;
 }
finally
 {
   System.out.println("Whoo-hoo!");
 }
```

Exceptions that occur in a catch or finally clause are handled normally; the search for an enclosing try/catch begins outside the offending try statement.

2.8.12 Summary

Exception handling is a mechanism that allows Java programs to handle various exceptional conditions, such as semantic violations of the language and program-defined errors, in a robust way. Exceptions are represented by instances of the class java.lang.Exception and its subclasses. Subclasses of Exception can hold specialized information (and possibly behavior) for different kinds of exceptional conditions. The try/catch guarding statements wrap a block of code and catch designated types of exceptions that occur within it. Well, if there is no enclosing try/catch statement, the exception pops to the top of the method in which it appeared and is, in turn, thrown from that method. In this way, the exception bubbles up until it's caught, or until it pops out of the top of the program, terminating it with a run-time error message. Java exceptions are divided into two categories: *checked exceptions* and *unchecked exceptions*. Most application level exceptions are checked, which means that any method that throws one, either by generating it itself (as we'll discuss below) or by passively ignoring one that occurs within it, must declare that it can throw that type of exception in a special throws clause in its method declaration. We can throw our own exceptions: either instances of Exception or one of its predefined subclasses, or our own specialized subclasses. You can also create your own exception classes by simply extending the Exception class. A finally clause can be added after a try and any associated catch clauses. Any statements in the body of the finally clause are guaranteed to be executed, no matter why control leaves the try body.

2.8.13 Review Questions

1. What is an exception? When is it generated?
2. How exceptions are handled in Java?
3. Explain with example how to create your own exceptions.
4. Differentiate between throw and throws clause.
5. What is the purpose of finally block?

2.8.14 Suggested Readings

1. The Complete Reference by Herbert Scheild, Mc-Graw Hill
2. Programming with Java by E.Balagurusamy, Mc-Graw Hill
3. Java : A Beginner's Guide by Herbert Schildt, Mc-Graw Hill
4. Introduction to Java Programming by Y.Daniel Cians, Prentice Hall
5. Object Oriented Programming in Java by G.T. Thampi
6. Java Programming by C. Xavier

Web Resources

www.tutorialspoint.com/java/
www.javapoint.com/java-tutorial
www.w3schools.in/java-tutorial/
www.programiz.com/java-programming

Print Setting by Department of Distance Education
Punjabi University, Patiala
2018

DEPARTMENT OF DISTANCE EDUCATION
PUNJABI UNIVERSITY, PATIALA
STUDENT'S RESPONSE-SHEET

Roll No.....
Class : BCA Part-III
Academic Session : 2018-2019
Date of receipt of lesson.....

Paper : BCA-302
Java Programming
Response Sheet No. 1 & 2
Marks obtained.....

Date & Signature of the Examiner.....

Date of submission of Response-Sheet by the student.....

No. of pages attached.....

Name & address of the student below in BLOCK LETTERS:
_____
_____
_____

Date of receipt in the Department.....

Long answer type questions. (Attempt any two) Each question carries 10 marks.

1. What is Java ? Explain the various features of Java.
2. What do you mean by Classes in Java ? Explain with the help of an example.
3. What is a Package ? What are the different categories of Package ? What are its advantages ?
4. What is an exception ? How is it generated ? How exceptions are handled in Java ?

2 x 10 = 20 marks

Short answer type questions. (Attempt any five) Each question carries 4 marks

1. Advantages of Java
2. Difference between Java and C++
3. Byte code
4. Inheritance
5. Constructors
6. Difference between static nested class and inner class
7. Method overriding
8. Abstract class

5 x 4 = 20 marks

Please send this Response-Sheet along with your answers to : The Deputy Registrar, Department of Distance Education, Punjabi University, Patiala-147002.