UNIT 3 CLASSES AND OBJECTS IN JAVA

Structure **Page Nos.**

3.0 INTRODUCTION

In Unit 2, we discussed the various datatypes, operators and keywords of Java. We also described the concept of mixing datatypes and type conversions. In addition, we also discussed the various programming construct used in Java and the method of using arrays in Java. In this Unit, we will discuss classes and objects, what constructors are and how they are used, creation of classes and subclasses, wrapper classes and inner classes, along with other useful concepts.

Classes are the single most important feature of Java. Everything in Java is either a class, a part of a class, or describes how a class behaves. 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. Both syntactically and logically, everything in Java happens inside a class. 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. For instance, consider the following Java programs:

```
class Hello World {

public static void main (String args[ ] ) {

System.out.printIn("Hello World");

 }

}

class GoodbyeWorld {

 public static void main (String args[ ] ) {

  System.out.printIn("Goodbye Cruel World!");
  }
}
```

Save this code in a single file called hellogoodbye.java, and compile it with the command javac hellogoodbye.java. Then list the contents of the directory. You will see that the compiler has produced two separate class files, HelloWorld.class and GoodbyeWorld.class.

The second class is a completely independent program. Type java GoodbyeWorld and then type java HelloWorld. These programs run and execute independently of each other although they exist in the same source code file.

3.1 OBJECTIVES

After going through this Unit, you will be able to:

- understand Constructor;
- understand Subclassing;
- understand Access Modifiers;
- understand Wrapper Classes;
- understand Inner Classes;
- understand the use of final keyword, and
- control access of variable and methods.

3.2 CLASSES AND OBJECTS

Classes are the single most important feature of Java. Everything in Java is either a class, a part of a class, or describes how a class behaves. Although classes will be covered in great detail in Section Four, they are so fundamental to an understanding of Java programs that a brief introduction is going to be given here.

All the action in Java programs takes place inside class blocks, in this case the HelloWorld class. 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. This may be a little confusing to C++ programmers who are used to defining all but the simplest methods outside the class block, but this approach is really more sensible. C++ takes the road it does primarily out of a desire to be compatible with C, not out of good object-oriented design. Both syntactically and logically everything in Java happens inside a class.

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, not source code files like in C. For instance consider the following Java program:

```
class HelloWorld {

public static void main (String args[ ]) {

System.out.printIn(Hello World);
  }

}

class GoodbyeWorld {

 public static void main (String args[ ]) {

 System.out.printIn(Goodbye Cruel World!);
  }
}
```

Save this code in a single file called hellogoodbye.java in your javahtml directory, and compile it with the command javac hellogoodbye.java. Then list the contents of the directory. You will see that the compiler has produced two separate class files, HelloWorld.class and Goodbye World.class.

The second class is a completely independent program. Type java GoodbyeWorld and then type java HelloWorld. These programs run and execute independently of each other although they exist in the same source code file. Off the top of my head I cannot think of why you might want two separate programs in the same file, but if you do the capability is there.

It is more likely that you will want more than one class in the same file. In fact, you will see source code files with many classes and methods.

3.3 CONSTRUCTOR

Constructors are methods defined inside a class which have the same name as the class, and which are used to create an instance of a class. This can also be called creating an object, as in:

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

Constructors are also used for the initialization of objects.

Constructors are identified by the two following rules:

- The method name must exactly match the class name
- There must not be a return type declared for the method.

Example:

```
public class ABC
{
        //member variable
        public ABC ()
        {

                //initialization code
    }
   //Constructor with arguments

   public ABC(int i)
   {

        //necessary code
   }
}
```

This example shows that we can also **overload** Constructor as any other method.

A Constructor with return type will be treated as a different method, and will not be called implicitly. The compiler, in some cases, for example, where you have defined a Constructor with a return type and call it to initialize an object, will flag an error stating **incorrect number of arguments.**

- The Default Constructor

Every class has at least one constructor. If you do not write any Constructor for your class, Java provides a Constructor with no arguments, and an empty body. This default Constructor allows you to create an object with the **new** operator by using the syntax **new ABC();**

If you have written a Constructor for your class, then in this case the default Constructor will not be available. However, if you still want to use it, you will have to write one yourself. For example, consider the following application in which the constructor assigns a string to an instance variable and then prints out the string.

```
Class Constructor Test

{

            String str;
            public Constructor Test (String str)
           {

                         this.str = str;
                         System.out.printIn(str);
           }

}

class TestingConstructor
{

         public static void main(String argv[ ])
        {

                        new ConstructorTest("Jack");
                        new ConstructorTest("Scott");
                        new ConstructorTest("Ted");
           }
}
```

At the console you do:

```
Javac TestingConstructor.java
Java TestingConstructor
```

The output of the above code is:

```
Jack
Scott
Ted
```

3.4 SUBCLASSING

The Java programming language is an Object Oriented language. It means that the language tries to map the real-life object into the system, or writing the code. The real world is heavily based on the system of sub-classification. That means that there exists a parent-child relation. The child inherits some of the properties of its parent, and at the same time also exhibits some of its own properties.

In object oriented programming, we often achieve this relationship (parent-child) though **inheritance**. Deriving a class from an existing class to inherit its traits in Java is called **Subclassing**.
Suppose we think of an example of a generic class, say Vehicle, that has all the properties and behaviours that are essential in any kind of a vehicle. Here is an example of such a class having some of the essential requirements:

```
abstract class Vehicle

{

              int capacity;
```

```
        Engine engine; //Engine is an another class
        abstract Public void move();
}
```

Note: This class is abstract, which means that it cannot be instantiated. The term implies that a programmer intentionally or by mistake cannot create an object of this abstract class. The concept is that we create a base class that incorporates within it the generic functionality of the real-world object that we are trying to mirrorize in the program. Hence, we make the base class abstract and therefore, make sure that the child classes derived from this class implement the overridden methods that have their generic form in the base class.

If you want some vehicle to exist, you need a more specialised version of the vehicle class. For example, you might want a model for a Car. Clearly, a Car actually is a Vehicle, but a Vehicle with additional features.

If we want to declare a Car class with some essential feature, the normal methodology is:

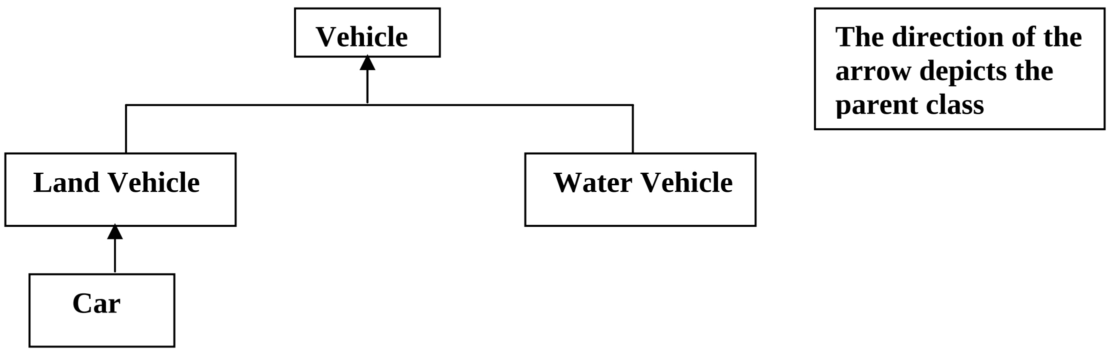
```
Class Car
{

        int capacity
        Engine engine; //Engine is an another class
        int wheels;
        Abstract Public void move();

}
```

The class Car is declared with minimum features and still you can clearly make out that the code has been duplicated. This leads to **code redundancy**, which in any case should be avoided. **Subclassing** is a way to create a new class from an existing class.

Clearly we can always say that a Car **is a** vehicle. This is nothing but **Inheritance.**



The above Figure shows the **Inheritance hierarchy.** Vehicle is the uppermost class. Land Vehicle inherits some properties of Vehicle, and Car inherits properties of both vehicle and Land Vehicle. Java supports only **Single Inheritance,** it does not support multiple inheritance. This means a class can have only one parent class and not more than one parent classes. Using it irresponsibly might result in ambiguities. Single inheritance makes code more reliable.

3.5 THE extends KEYWORD

From the previous example of Vehicle and Car, it is very clear that we want to declare a class in terms of another class. In terms of programming, inheritance is achieved through the **extends** keyword.

```
abstract class Vehicle
{
             int capacity;
            Engine engine; //Engine is an another class
            abstract public void move();
}

class Land Vehicle extends Vehicle
{

              public void move()
              {

                                   //code for the move function
              }
}
```

All other variables and methods in Car class are inherited from Vehicle and LandVehicle. This approach is a good improvement in terms of maintenance and reliability. If a correction is made in Vehicle class, then the Car class is corrected automatically without the programmer having to do anything, except compile it.

- **The Super Keyword**

The **super** keyword refers to the superclass (parent class) of the class in which the keyword is used. It is used to refer to the member variables or the methods of the superclass.

```
Class Base
{

             public void print()
             {

                       System.out.printIn(“”I am Superclass”);
             }

}

public class Derived extends Base
{

            public void printSuper()
           {

                       super.print();//call print() function of superclass (Base)
           }
           public void print()
          {

                       System.out.printIn(“I am Derived class”);
          }
          public static void main (String s[ ])
         {

                       Derived d=new Derived ();
                       d.printSuper();
                       d.print();
}
```

At the console, you do:

```
    Javac Derived.java
    Java Derived
```

get the output:

```
    I am Superclass
    I am Derived class
```

3.6 THE instance of OPERATOR

In object oriented programming language, **upcasting** is always legal. It implies that a child class object can be referenced through a parent class reference variable. If the objects are passed around say, in a function using reference to their parents, the **instanceof** operator provides a way to differentiate between them as it is shown in example:

```
class Student

{

        public void print()

        {

                System.out.printin("I am a student");
        }

}
class Firstyear_stu  extends Student

{

        public void print()

       {
               System.out.printin("I am a first year student");

       }

}

class Secondyear_stu extends Student
{
      public void print()
      {

              System.out.printin("I am a second year student");
      }

}

public class Identification
{

       public void identify (Students)
       {

                if(s instance of Firstyear_stu)
                {
```

```
                //*here any decision depending up on the type of
                object can be taken*/
                s.print();
                System.out.printin("Assign roll no in the range 1000 to
            2000");
            }
            if(s instance of Secondyear_stu)
            {
                s.print();
                System.out.printin("Assign roll no in the range 2000 to
                                                                3000");
            }

}

public static void main(String s[ ])

{

            Firstyear_stu fs=new Firstyear_stu();
            Seccondyear_stu ss=new Secondyear_stu();
            Identification id=new Identification();
            id.identify(fs);
            id.identify(ss);
 }

}
```

At the console, you do:

```
Javac Identifcation.java
Java Identification

I am a first year student
Assign roll no in the range 1000 to 2000
I am a second year student
Assign roll no in the range 2000 to 3000
```

3.7 STATIC VARIABLES AND METHODS

Sometimes, it is desirable to have a variable that is shared among all instances of a class. This can be used for the communication between objects of a class, or for taking some global decision.
Example:

```
Public class Global
{
    public static int i=0;
    public Global()
    {
        i++;
    }
    public void print()
    {

        System.out.printin("I am object no " +I+" of class
        Global");
    }
    public static void main(String s[ ])
    {
            for(int  j=0; j<5;j++)
            {
                    new Global().print();

            }

        }

}
```

The output of the above code would be:

```
I am Object no 1 of class Global
I am Object no 2 of class Global
I am Object no 3 of class Global
I am Object no 4 of class Global
I am Object no 5 of class Global
```

Sometimes you want to access program code without instantiating the class to which that code belongs. A method or variable, marked as static, can be used for this purpose.

Example:

```
Class ExStatic
{
         static int i=10;
         public static void print_i()
        {

             System.out.printIn("Value of i is "+i);
        }
        public static void main(String s[ ])
        {
             ExStatic.print_i();//method is called using class name
        }

}
```

At the console, you do:

```
javac ExStatic.java
java ExStatic

Value of i is 10
```

Since the static method can be called without creating any object of a class, **this** reference is not passed in it. This is why it cannot access non-static variables.

Note: **main** method is static because it has to be accessible before any instance can take place.

3.8 THE final KEYWORD

The final modifier applies to classes, methods, and variables. The meaning of **final** varies from context to context, but the essential idea is the same: final features may not be changed.

A **final class** cannot be subclassed. For example, the code below will not compile, because the java.lang.String class is final.

```
                Class SubString extends String {}
```

Compiler will give an error '**can't subclass final classes**'. The use of final classes is where the class defined is so specific that it should not be modified through an extension.

A **final variable** may not be modified once it has been assigned to a value. In Java, final variables play the same role as **const** variables in C++ and **#define** constants in C.

A **final method** cannot be overridden. For example, if the following code was compiled, an error will be generated.

```
Public class ErrorTester
{
            public static void main(String argv[ ])
            {

                    Polygon poly = new Polygon ( );
                    Poly.draw();
                    Rectangle rect = new Rectangle ();
                    rect.draw();

             }
}

class Polygon
{
         public final void draw ()
         {

                  System.out.println("Draw in polygon");

          }

}
class Rectangle extends Polygon
{

          public void draw()
          {

                  System.out.println("Draw in Rectangle");
           }
}
```

3.9 ACCESS CONTROL

You can see that the terms super, public and protected are used in previous programs. Can you tell what is the role of these terms in programs? Right, public and protected are access controller, used to control the access to members (data members and member functions) of a class, and super is used in implementing inheritance.

Now, you can see how access control is used in Java programs.

Controlling Access to Members of a Class

One of the objectives of having access control is that classes can protect their member data and methods from being accessed by other objects. Why is this important? Well, consider this. You are writing a class that represents a query on a database that contains all kinds of secret information; say student records, or marks obtained by a student in a final examination.

In your program, you will have certain information and queries contained in the class. Class will have some publicly accessible methods and variables in your query object, and you may have some other queries contained in the class simply for the personal use of the class. These methods support the operation of the class, but should not be used by objects of another type. In other words you can say–you have secret information to protect.

How can you protect it?

In Java, you can use access specifiers to protect both variables and methods of a class when you declare them. Java language supports four distinct access specifiers for member data and methods: private, protected, public, and if left unspecified, package.

The following chart shows the access level permitted by each specifier.

Specifier	**Class**	**subclass**	**package**	**world**
Private	X			
Protected	X	X*	X	
Public	X	X	X	X
Package	X		X	

The first column indicates whether the class itself has access to the members defined by the access specifier. As you can see, a class always has access to its own members.

The second column indicates whether subclasses of the class (regardless of which package they are in) have access to the member.

The third column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member.
The fourth column indicates whether all classes have access to the member.

Note that the protected/subclass intersection has an '*'. This particular case has a special association with inheritance implementation. You will see in the next section of this unit, how protected specifier is used in inheritance. Package will be covered in the next unit of this Block.

☞ Check Your Progress 1

1) What is the advantage of inheritance? How can a class inherit the properties of any other class in Java?

……………………………………………………………………………………

……………………………………………………………………………………

2) Explain the need of access specifiers.

…………………………………………………………………………………

…………………………………………………………………………………

……

3) When is private specifier used in a program?

……………………………………………………………………………………

……………………………………………………………………………………

……

Let’s look at each access level in more detail.

Private

Private is the most restrictive access level. A private member is accessible only to the class in which it is defined. You should use this access to declare members that you are going to use within the class only. This includes variables that contain information which, accessed by an outsider could put the object in an inconsistent state; or methods, if invoked by an outsider, could jeopardize the state of the object or the program in which it is running. You can look at private members like secrets you never tell anybody.

To declare a private member, use the `private` keyword in its declaration. The following class contains one private member variable and one private method:

```
class First
{
private int MyPrivate; // private data member
private void privateMethod() // private member function
{
System.out.println("Inside privateMethod");
}
}
```

Objects of class First can access or modify the MyP`rivate` variable and can invoke `privateMethod.O`bjects other than class First cannot access or modify MyP`rivate` variable, and cannot invoke `privateMethod` . For example, the Second class defined here:

```
class Second {
void accessMethod() {
First a = new First();
a. MyPrivate = 51;      // illegal
a.privateMethod();      // illegal
}
}
```

cannot access the MyPrivate variable or invoke `privateMethod` of the object of First.

If you are attempting to access a method to which it does not have access in your program, you will see a compiler error like this:

```
Second.java:12: No method matching privateMethod()
found in class First.
a.privateMethod();        // illegal
1 error
```

A very interesting question that can be asked is whether one object of class First can access the private members of another object of class First. The answer to this question is given by the following example. Suppose the First class contained an instance method that compared the current First object (`this`) to another object based on their `iamprivate` variables:

```
class Alpha
{
private int MyPrivate;
boolean isEqualTo (First anotherObject)
{
if (this. MyPrivate == anotherobject. MyPrivate)
return true;
else
return false;
}
}
```

This is perfectly legal. Objects of the same type have access to one another's private members. This is because access restrictions apply at the class, or type level (all instances of a class) rather than at the object level.

Now, let us discuss protected specifier.

Protected

Protected specifiers allows the class itself, subclasses, and all classes in the same package to access the members. You should use the protected access level for those data members or member functions of a class which you can be accessed by subclasses of that class, but not unrelated classes. You can see protected members as family secrets – you do not mind if the whole family knows, and even a few trusted friends, but you would not want any outsiders to know. A member can be declared protected using keyword `protected`.

```
public class Student
{
protected int age;
public  String name;
protected void protectedMethod()
{
System.out.println("protectedMethod");
}
}
```

You will see the use of protected specifier in programs discussed in the next sections of this block.

Public

This is the easiest access specifier. Any class, in any package, can access the public members of a class. Declare public members only if you want to provide access to a member by every class. In other words, you can say if access to a member by an outsider cannot produce undesirable results, the member may be declared public. To declare a public member, use the keyword `public`. For example,

```
public class Account
{
public String name;
protected String Address;
protected int Acc_No;
public void publicMethod()
{
System.out.println("publicMethod");
}
}
class Saving_Account
{
void accessMethod()
{
Account a = new Account();
String MyName;
a.name =  MyName;       // legal
a.publicMethod();       // legal
}
}
```

As you can see from the above code snippet, Saving_Account can legally inspect and modify the `name` variable in the `Account class` and can legally invoke `publicMethod` `also`.

Member Access and Inheritance

Now we will discuss uses of the **super** keyword in Java programming.

There are two uses of the super keyword.

1. It is used for calling the superclass constructor.
2. It is used to access those members of superclass that are hidden by the member of subclass.

How a subclass can hide member of a superclass?

Can you tell why a subclass is called constructor of superclass?

An Object of class is created by a call constructor to initialize its data member! Now, if you create an object of a subclass you will call a suitable constructor of that subclass to initialize its data members. Can you tell how those data members of the parent class, which a subclass is inheriting will be initialized? Therefore, to initialize superclass (parent class) data member, a superclass constructor is called in the subclass constructor.

To call a superclass constructor, write super **(argument-list)** in subclass constructor, and this should be the very first statement in the subclass constructor. This argument list includes the arguments needed by superclass constructor. Since the constructor can be overloaded, super () can be called using any form defined by the superclass. In case of constructor overloading in superclass, which of the constructors will be called is decided by the number of parameters or the type of parameter passed in super( ).

Now, let us take one example program to show how subclass constructor calls superclass constructor.

```
class  Student
{
String name;
String address;
int age;
Student( String a, String b, int c)
{
name = a;
address = b;
age = c;
}
void display( )
{
System.out.println("*** Student Information ***");
Sstem.out.println("Name : "+name+"\n"+"Address:"+address+"\n"+"Age:"+age);
}
}
class PG_Student extends Student
{
int age;
int  percentage;
String course;
PG_Student(String a, String b, String c, int d , int e)
{
super(a,b,d);
course = c;
percentage = e;
age = super.age;
}
void display()
{
super.display();
System.out.println("Course:"+course);
}
}
class Test_Student
{
public static void main(String[] args)
{
 Student std1 = new Student("Mr. Amit Kumar" , "B-34/2 Saket J Block",23);
PG_Student pgstd1 = new  PG_Student("Mr.Ramjeet ", "241- Near Fast Lane Road
Raipur" ,"MCA", 23, 80);
std1.display();
pgstd1.display();
}
}
```

Output:

```
*** Student Information ***
Name : Mr. Amit Kumar
Address:B-34/2 Saket J Block
Age:23
*** Student Information ***
Name : Mr.Ramjeet
Address:241- Near Fast Lane Road Raipur
Age:23
Course:MCA
```

In the above program, PG_Student class is derived from Student class. PG_Student class constructor has called constructor of Student class. One interesting point to note in this program is that both Student and PG_Student classes have a variable named age. When PG_Student class will inherit class Student, member data **age** of Student class will be hidden by the member data **age** of PG_Student class. To access member data **age** of Student class in PG_Student class, **super.age** is used. Whenever any member of a superclass has the same name of the member of subclass, it has to be accessed by using the super keyword prefix to it.

3.10 METHOD OVERRIDING

You know that a subclass extending the parent class has access to all the non-private data members, and methods its parent class. Most of the time, the purpose of inheriting properties from the parent class and adding new methods is to extend the behaviour of the parent class. However, sometimes it is required to modify the behaviour of the parent class. To modify the behaviour of the parent class overriding is used.

Some important points that must be taken care while overriding a method:

(i) An overriding method (largely) replaces the method it overrides.
(ii) Each method in a parent class can be overridden, at most, once in any one of the subclass.
(iii) Overriding methods must have exactly the same argument lists, both in type and in order.
(iv) An overriding method must have exactly the same return type as the method
it overrides.
(v) Overriding is associated with inheritance.

The following sample program shows how member function area () of the class Figure is overridden in subclasses Rectangle and Square.

```
class Figure
{
double sidea;
double sideb;
Figure(double a, double b)
{
sidea = a;
sideb = b;
}
Figure(double a)
{
sidea = a;
sideb = a;
}
double area( )
{
System.out.println("Area inside figure is Undefined.");
return 0;
}
}
class Rectangle extends Figure
{
Rectangle( double a , double b)
{
super ( a, b);
}
double area ( )
{
System.out.println("The Area of Rectangle:");
return sidea*sideb;
}
}
class Squre extends Figure
{
Squre( double a )
{
super (a);
}
double area( )
{
System.out.println("Area of Squre: ");
return sidea*sidea;
}
}
class Area_Overrid
{
public static void main(String[] args)
{
Figure f = new  Figure(20.9, 67.9);
Rectangle r = new  Rectangle( 34.2, 56.3);
Squre s = new Squre( 23.1);
System.out.println("***** Welcome to Override Demo ******");
f.area();
System.out.println(" "+r.area());
System.out.println(" "+s.area());
}
}
```

Output:

```
***** Welcome to Override Demo ******
Area inside figure is Undefined.
The Area of Rectangle:
 1925.46
Area of Squre:
 533.61
```

In most of the object oriented programming languages like C++ and Java, a reference parent class object can be used as reference to the objects of derived classes. In the above program, to show the overriding feature, object of *Figure class* can be used as reference to objects of Rectangle and Square class. Above program with modification (shown in bold) in Area_Override class will be as :

```
class Area_Override
{
public static void main(String[] args)
{
Figure f = new  Figure(20.9, 67.9);
Rectangle r = new  Rectangle( 34.2, 56.3);
Squre s = new Square( 23.1);
System.out.println("***** Welcome to Override Demo ******");
f.area();
f= r;
System.out.println(" "+f.area());
f = s;
System.out.println(" "+f.area());
}
}
```

☞ Check Your Progress 2

1) State True or False for the following statements:

	T	F
(i) One object can access the private member of the object of the same class.	☐	
(ii) A subclass cannot call the constructor of its super class.	☐	
(iii) A public variable in a package cannot be accessed from other package.	☐	

2) Explain the use of the super keyword in Java programming.

…………………………………………………………………………………

…………………………………………………………………………………

……

…………………………………………………………………………………
…

3) How is method overriding implemented in Java? Write the advantage of method overriding.

…………………………………………………………………………………

…………………………………………………………………………………

………………………………………………………………………………………
………

3.11 ABSTRACT CLASSES

As seen from the previous examples, when we extend an existing class, we have a choice whether to redefine the methods of the superclass. Basically, a superclass has common features that are shared by subclasses. In some cases, you will find that a superclass cannot have any instance (object) and those of such classes are called **abstract classes**. Abstract classes usually contain **abstract methods.** Abstract method is a method signature (declaration) without implementation. Basically, these abstract methods provide a common interface with different derived classes. Abstract classes are generally used to provide common interface derived classes. You know a superclass is more general than its subclass(es). The superclass contains elements and properties common to all of the subclasses. Often, the superclass will be set up as an *abstract* class, which does not allow objects of its prototype to be created. In this case, only objects of the subclass are created. To do this, the reserved word *abstract* is included (prefixed) in the class definition.

For example, the class given below is an abstract class.

```
public abstract class Player  // class is abstract
{
private String name;
public Player(String  vname)
{
name=vname;
}
public String getName()  // regular method
{
return (name);
}
public abstract void Play();
//  abstract method: no implementation
}
```

Subclasses *must* provide the method implementation for their particular meaning. If the method statement is one provided by the superclass, it would require overriding in each subclass. In case you forget to override, the applied method statement may be inappropriate.

Now can you think what to do in case for derived classes must redefine the methods of superclass?

The answer is very simple Make those methods abstract.

In case attempts are made to create objects of abstract classes, the compiler generates an error message. If you are inheriting in a new class from an abstract class and you want to create objects of this new class, you must provide definitions to all the abstract methods in the superclass. If all the abstract methods of super class are not defined in this new class this class also will become abstract.

Is it possible to have an abstract class without an abstract method? Yes, you can. Can you think about the use of such abstract classes? These types of classes are defined when it does not make any sense to have any abstract methods in the class and yet, you want to prevent an instance of that class.

Inheritance represent, “is–a” relationship between a subclass and a superclass. In other words, you can say that every object of a subclass is also a superclass object with some additional properties. Therefore, the possibility of using a subclass object in place of a superclass object is always there. This concept is very helpful in implementing polymorphism.

Now we will discuss polymorphism, one of the very important features of object oriented programming, called polymorphism supported by Java programming language.

3.12 POLYMORPHISM

Polymorphism is the capability of a *method* to do different things based on the object through which it is invoked, or the object it is acting upon. For example, method *find _area* will work definitely for Circle object and Triangle object In Java, the type of actual object always determines method calls; object reference type does not play any role in it. You have already used two types of polymorphism (overloading and overriding) in the previous unit and in the current unit of this block. Now we will look at the third: *dynamic method binding*. Java uses the Dynamic Method Dispatch mechanism to decide at run time which overridden function will be invoked. Dynamic Method Dispatch mechanism is important because it is used to implement runtime polymorphism in Java. Java uses the principle: “a super class object can refer to a subclass object” to resolve calls to overridden methods at run time.

If a superclass has a method that is overridden by its subclasses, then the different versions of the overridden methods are invoked or executed with the help of a superclass reference variable.
Assume that three subclasses (Cricket_Player, Hockey_Player and Football_Player) that derive from Player abstract class are defined with each subclass having its own Play() method.

```
abstract class Player  // class is abstract
{
private String name;
public Player(String nm)
{
name=nm;
}
public String getName()        // regular method
{
return (name);
}
public abstract void Play();
//  abstract method: no implementation
}
class Cricket_Player extends Player
{
Cricket_Player( String  var)
{
}
public void Play()
{
System.out.println("Play Cricket:"+getName());
}
}
class Hockey_Player extends Player
{
Hockey_Player( String  var)
{
}
public void Play()
{
System.out.println("Play Hockey:"+getName());
}
}
class Football_Player extends Player
{
Football_Player( String  var)
{
}
public void Play()
{
System.out.println("Play Football:"+getName());
}
}
public class PolyDemo
{
public static void main(String[] args)
{
Player ref;  // set up var for an Playerl
Cricket_Player  aCplayer = new Cricket_Player("Sachin");  // makes specific objects
Hockey_Player aHplayer = new Hockey_Player("Dhanaraj");
Football_Player aFplayer = new Football_Player("Bhutia");
// now reference each as an Animal
ref = aCplayer;
ref.Play();
ref = aHplayer;
ref.Play();
ref = aFplayer;
ref.Play();
}
}
```

Output:

```
Play Cricket:Sachin
Play Hockey:Dhanaraj
Play Football:Bhutia
```

Notice that although each method is invoked through ref, which is a reference to player class (although no player objects exist), the program is able to resolve the correct method related to the subclass object at runtime. This is known as dynamic (or late) method binding.

3.13 WRAPPER CLASSES

In Java, primitive data types such as long, int, char etc. are not treated as objects. This is for simplicity and efficiency; but many times, you need to treat these primitive type as objects, for example, when you want to store primitive data types along with other objects in an array. The Java programming language provides **"Wrapper"** to manipulate primitive data elements as objects. Each primitive data type has a corresponding wrapper class in the java.lang package. These classes are:

Primitive Data Type	**Wrapper class**
Boolean	Boolean
Byte	Byte
Char	Character
Short	Short
Int	Integer
Long	Long
Float	Float

Double	Double

A wrapper class object is constructed by passing the value to be wrapped into the appropriate constructor. For example:

```
Int ptyp=10;
Integer pobj = new Integer(ptyp);
```

Note: Wrapper classes also provide some useful methods.

3.14 INNER CLASSES

A class that is declared and defined inside some other class is called an **inner class.** Sometimes, it is also called a nested class. The inner classes give additional functionality to the program, and make it clearer. Let us look at the basic construct of the inner class.

```
Public class Outer
{

         int i=10;
         public class Inner
        {

                 int j=20;
                 public void innerFn()
                {
                System.out.printIn("j is "+ j);
                }

        }
       public void outerFn()
       {
        System.out.printIn("I is "+I);
        }

       public static void main(String s [ ])
       {
               Outer out =new Outer();
               out.outerFn();
       }

}
```

This example consists of a class **Outer,** which includes an inner class **Inner**. The compiler generates a class file, **Outer$Inner.class** in addition to **Outer.class.**

Enclosing the 'this' reference of Inner Classes.

Inner classes have access to their enclosing class's scope. The access to enclosing class's scope is possible because the **inner class** actually has a hidden reference to the outer class **this reference.**

Example:

```
public class Outer
{

         int i=10;
```

```
        public class Inner
        {

                int j;
                public void innerFn()
                {
                       System.out.printIn("I is "+I);
                      System.out.printIn("j is "+j);
                 }
         }
         public void innerCreate()
        {
                Inner in=new Inner();
                in.innerFn();

         }

        public void outerFn()
        {
                System.oaut.printIn("I is "+ i);
         }

         public static void main(String s[ ] )
         {
            Out out=new Outer();
            out.innerCreate();
         }
  }
```

The above example shows that an inner class can access the member of it is outer class. This is possible only because it has a hidden reference to outer class.

At the console, you do:

```
JavacOuter.java
JavaOuter

i is 10
j is 0
```

Inside the method **innerCreate** an instance of inner class **Inner** is created and a method **innerFN()** is called.

☞ Check Your Progress 3

1) Define a class and an object. How do you create a class? How do you declare an object? How do you create an object? How do you declare and create an object in one statement?

……………………………………………………………………………………...

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

2) What is a class method?

……………………………………………………………………………………...

………………………………………………………………………………………

…………………………………………………………………………………

……………………………………………………………………………….

3.15 SUMMARY

In this Unit, you have learned mainly about class and object concepts, the role of constructor inheritance concept, static variables and access control mechanism.
A class is a fundamental unit in java. Everything is encapsulate in a class itself. An object is an instance of a class. A Java program can be thought of a large set of cooperating objects.

Constructors are special methods that initialize an objects data. It has exactly the same name as the class it comes from. It can be overloaded making it easier to construct objects with different kinds of initial data values. Constructors do not require a return type, not even void.

3.16 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Inheritance is used for providing reusability of pre-existing codes. For example, there is an existing class A, and you need another class B which has class A properties, as well as some additional properties. In this situation, class B may inherit from class A, and there is no need to redefine the properties common to class A and class B. In Java, a class is inherited by any other class using *extends* keyword.

2) Access specifiers are used to control the accessibility of data members and member functions of a class. It helps classes to prevent unwanted exposure of members (data and functions) to the outside world.

3) If some data members of a class are used in internal operations only then there is no need to provide access of these members to outside world. Such member data should be declared private. Similarly, those member functions which are used for internal communications/operations only should be declared private.

Check Your Progress 2

1)
   i. True
   ii. False
   iii. False

2) Java's super keyword is used for two purposes.

   i. To call the constructors of the immediate superclass.
   ii. To access the members of the immediate superclass.

When a constructor is defined in any subclass, it needs to initialize its superclass variables. A subclass can call a constructor method defined by its superclass by use of following form of ***super (parameter_list).*** Parameters needed by the superclass constructor are passed in ***super (parameter_list)***. The super keyword helps in conflict resolution in subclasses in the situation of "when members name in superclass is the *same as* members name in a subclass, and the members of the superclass is to be called in subclass".
super.member; // The member may be either member function, or member data

3) Java uses Dynamic Method Dispatch, one of its powerful concepts to implement method overriding. Dynamic Method Dispatch helps in deciding the

version of the method to be executed. In other words, to identify the type of object on which method is invoked.

Overriding helps in:

- Redefining inherited methods in subclasses. In redefinition, the declaration should be identical, while the code may be different. It is like having another version of the same product.
- Adding more functionality to a method.
- Representing an abstract concept, i.e, abstract class. In this case, it becomes essential to override methods in a derived class of abstract class.

Check Your Progress 3

1) A class is a template for what an objects data and method will be. An object is an instance of a class. You can create many instances of a class creating an instance is referred to as instantiation. In order to declare an object you must declare a class variable that represents that object.

The syntax for declaring a class is

c**lass employee**

The syntax for declaring an object is

engineer=new employee();

The following is an example of creating and instantiating an engineer object in one step.

employee engineer = new employee();

2) A method that does not use the instance variable can be defined as a class method. The method can be invoked without creating an object of the class.