COMPUTER SCIENCE

B. Sc. II (CBCS) Semester-IV 2023-2024

2CS2 : RDBMS and Core Java

Unit-V : Class & Inheritance

PROF. V. V. AGARKAR

Assistant Professor & Head

Department of Computer Science

Shri. D. M. Burungale Science & Arts College, Shegaon, Dist. Buldana

Unit – V

***Syllabus*: Classes & OOPs:** Class, Object, Method, Constructor: types, this Keyword, **Polymorphism**: Overloading & Overriding, **Inheritance**: types of inheritance, Super, Abstract class, **Interfaces**: Interface concept, Defining, and Implementing of Interface, Using Final (variables, methods and classes), Garbage Collection.

Introduction

Java is a true object-oriented language and therefore the underlying structure of all Java programs is classes. Anything we wish to represent in a Java program must be encapsulated in a class that defines the *state* and *behaviour* of the basic program components known as *objects*. Classes create objects and objects use methods to communicate between them.

Class

The class is at the core of Java. It is the logical construct upon which the entire Java program is built because it defines the shape and nature of an object. Any concept you wish to implement in a Java program must be encapsulated within a class.

A class defines a new (user defined) data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a *template* for an object, and an object is an *instance* of a class.

Class represents the set of properties or methods that are common to all objects of that class type. Classes provide a convenient method for packing together a group of logically related data items and functions that work on them.

In Java, the data items are called *fields* or *instance–variables* and the functions are called *methods*.

- ***Defining Class***

When you define a class, you declare its exact form and nature. You do this by specifying the data (called data fields or instance-variables) that it contains and the code (called methods or member functions) that operates on that data. While very simple classes may contain only code or only data, most of the classes contain both. The basic form of a class definition is:

```
class classname [extends superclassname]
{
        [ fields declaration; ]
        [ methods declaration; ]
}
```

A class is declared by use of the **class** keyword. Classes usually consist of two things: instance variables and methods. A more detailed form of class declaration is:

```
class classname
{
    type instance-variable1;
    type instance-variable2;
    ..........
    type instance-variableN;

    type methodname1(parameter-list)
    {
        // body of method
    }
    type methodname2(parameter-list)
    {
        // body of method
    }
    ...
    type methodnameN(parameter-list)
    {
        // body of method
    }
}
```

Data and code that operates on the data are encapsulated in a class by placing data fields and methods inside the body of the class definition. Collectively, the methods and variables defined within a class are called ***members*** of the class. The data fields or variables defined within a class are called ***instance variables*** because they are created whenever an object of the class is instantiated. The instance variables can be declared exactly the same way as that local variables are declared. Each instance of the class (*i.e.* each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. Instance variables are also known as *member variables*.

The code defined within a class is contained within ***methods***. In most classes, the instance variables are accessed by the methods defined for that class. Thus, as a general rule, it is the methods that determine how a class data can be used.

Example-1:

Let us consider a simple class called **Rectangle** that defines two instance variables: **width** and **length**. Here, **Rectangle** does not contain any methods.

```
class Rectangle
{
    int length;
    int width;
}
```

Above class defines a new type of data. In this case, the new data type is called **Rectangle**. You will use this name to declare objects of type **Rectangle**. It is important to remember that a **class** declaration only creates a template; it does not create an actual object.

Objects

When a class is created, actually a new data type is created. This new data type is used to declare objects of that type. An object in Java is essentially a block of memory that contains space to store all the instance variables.

• *Creating Objects*

Creating an object is also referred to as ***instantiating*** an object. However, creating objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the `new` operator.

The `new` operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is the address in memory of the object allocated by `new`. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

• *Declaring objects*

Objects can be declared by using following syntax:

```
Classname  objectname;
```

This declares a variable to hold the object reference. It just creates a reference and it does not point to an actual object.

Example-2:

1) `Rectangle rect1;`
2) `Box mybox;`

The first example declares `rect1` as a reference to an object of type `Rectangle` and second example declares `mybox` as a reference to an object of type `Box`.

After the execution of the statements in example 1 and 2, `rect1` and `mybox` contains the value null, which indicates that it does not yet point to an actual object. Any attempt to use `rect1` and `mybox` at this point will result in a compile-time error.

• *Creating (or Instantiation) objects*

Objects can be created by using following syntax:

```
Objectname  =  new  Classname();
```

Objects in Java are created using the `new` operator. The `new` operator creates an object of the specified class and returns a reference to that object. This allocates an actual object and assigns object reference to the variable.

Example-3:

1) `rect1 = new Rectangle();`
2) `mybox = new Box();`

After the execution of the statements in example 1 and 2, `rect1` and `mybox` acquire an actual, physical copy of the objects and assign it to the variables `rect1` and `mybox` respectively. After this, `rect1` and `mybox` can be used as objects of `Rectangle` and `Box` classes respectively.

The statement of declaration and creation of objects can be combined into one statement as follows:

```
Classname  Objectname  =  new  Classname();
```

Example-4:

The above Example-2 and 3 can be rewritten into a single statement as follows:

1) `Rectangle rect1 = new Rectangle();`
2) `Box mybox = new Box();`

It is noted that any number of object can be created for a class. Each object has its own copy of the instance variables of its class. This means that any changes to the variables of one object have no effect on the variables of another.

Example-5:

```
Rectangle rect1 = new Rectangle();
Rectangle rect2 = new Rectangle();
Rectangle rect3 = new Rectangle();
```

Methods

Methods must be added to the class for manipulation of the data contained in the class. Methods are used to access the instance variables defined by the class. Methods are declared inside the body of the class but immediately after the declaration of instance variables. In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself. General form of a method declaration is:

```
type methodname (parameter-list)
{
     method-body;
}
```

Method declarations have four basic parts:

- The name of the method (*methodname*)
- The type of the value the method returns (*type*)
- A list of parameters (*parameter-list*)
- The body of the method (*method-body*)

The ***type*** specifies the type of value returned by the method. This can be a simple data type such as **int** as well as any class type. It can even be **void** type, if the method does not return any value. The ***methodname*** is a valid identifier other than those already used by other items within the current scope. The ***parameter-list*** is always enclosed in parentheses. This list contains a sequence of data type and variable pairs separated by commas. Parameters are essentially variables that receive the value of the *arguments* passed to the method when it is called. If the method has no parameters, then the parameter list will be empty and retains empty parenthesis. The body of the method actually describes the operations to be performed on the data. Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

```
return (value);
```

Here, *value* is the value returned.

Example-6:

Following are the examples of different method declarations.

1) `void getdata(int x, int y)`
2) `void getdata(int m, float y, float z)`
3) `void volume()`
4) `double volume()`
5) `int fact(int n)`

In the first example `getdata()` function accepts two parameters `x`, and `y` of the type `int` and return type is `void` that means it does not returns any value. In the second example `getdata()` function accepts three parameters `m`, `y` and `z` of the type `int`, `float` and `float` respectively and return type is `void` that means it does not returns any value. In the third example `volume()` function accepts no parameter and return type is `void` that means it does not returns any value. In the forth example `volume()` function accepts no parameter and return type is `double` that means it returns a value of type `double`. And in the fifth example `fact()` function accepts one parameter of type `int` and return type is `int` that means it returns a value of type `int`.

Example-7:

Let us consider the **Rectangle** class declared in ***Example-1*** again and add a method **getData( )** to it to calculate area of the box as follows:

```
class Rectangle
{
   int length;
   int width;
   void getData(int x, int y)
   {
         length = x;
         width = y;
   }
}
```

Note that the method **`getData()`** has a return type of **void** because it does not return any value. Two *int* values are passed to the method which are then assigned to the instance variables `length` and `width`. The **`getData()`** method is basically added to provide values to the instance variables.

Example-8:

Let us add one more method to the class **Rectangle** to compute the area of the rectangle defined by the class. This can be done as follows:

```
class Rectangle
{
   int length, width;
   void getData(int x, int y)
   {
         length = x;
         width = y;
   }
   int rectArea()
   {
         int area = length * width;
         return(area);
   }
}
```

The new method **`rectArea()`** computes area of the rectangle and returns the result, Since the result would be an integer, the return type of the method has been specified as `int`. Also note that the parameter list is empty.

- ***Accessing Class Members***

Once the objects are created, each object contains its own set of variables; values should be assigned to these variables in order to use them in the program.

All variables must be assigned values before they are used. From outside the class, the instance variables and the methods cannot access directly. To do this, the concerned object and the dot operator (`.`) must be used as shown below:

```
Objectname.variablename =  value;
objectname.methodname(parameter-list);
```

Here `objectname` is the name of the object, `variablename` is the name of the instance variable inside the object that want to access, `methodname` is the method that want to call, and `parameter-list` is a comma separated list of "actual values" that must match in type and number with the parameter list of the method declared in the class. The instance variables of the **Rectangle** class may be accessed and assigned values as follows:

```
rect1.length =  15;
rect1.width  =  10;
rect2.legth  =  20;
rect2.width  =  12;
```

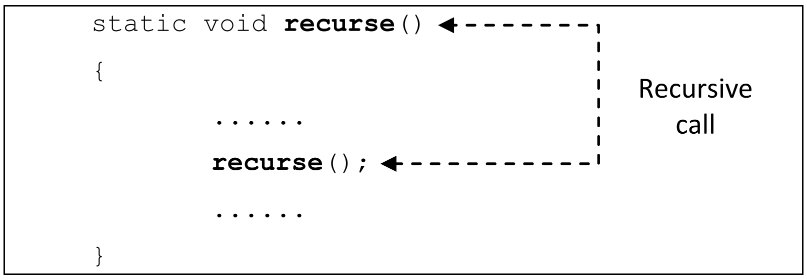
This is one way of assigning values to the variables in the objects. Another way and more convenient way of assigning values to the instance variables are to use a method that is, declared inside the class.

```
Rectangle rect1 = new Rectangle();   // Creating an object
Rect1.getData(15, 10};               // Calling the method using the object
```

This code creates **rect1** object and then passes in the values 15 and 10 for the **x** and **y** parameters of the method **getData**. This method then assigns these values to **length** and **width** variables respectively.

Recursion

Recursion is the process of defining something in terms of itself. Java also supports recursion. In Java, ***Recursion*** is a process in which a method calls itself directly or indirectly repeatedly to solve a problem. Such method is called a *recursive method*. The recursion provides a way to break complicated problems down into simple problems which are easier to solve.



```
static void recurse()
{
        ......
        recurse();
        ......
}
```

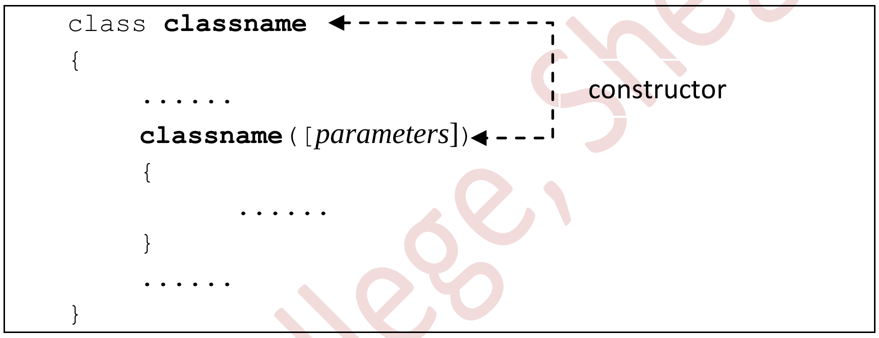
Example-9:

Following is recursive method `factorial()` which calculate factorial of *n*.

```
static int factorial(int n)
{
      if (n == 0)
            return 1;
      else
            return(n * factorial(n-1));
}
```

Constructors

A constructor is a special method used to initialize an object immediately upon creation. It has the same name as the class name in which it resides and is syntactically similar to method. Once it is defined, the constructor is automatically called immediately after the object is created, and before the `new` operator is completed. Every time an object is created using the `new` keyword, at least one constructor is called. The constructor never returns any value, not even *void*.



```
class classname
{
      ......
      classname([parameters])
      {
            ......
      }
      ......
}
```

- **Types of constructor in Java**

  There are two types of constructor in Java, which are

  1. Default constructor
  2. Parameterized constructor

1. Default constructor

A constructor that has no parameters is known as ***default constructor***.

If we don't define any constructor in a class, the compiler automatically builds default constructor for the class. It initializes the object with default values (e.g., null for reference types, 0 for numeric types, false for Boolean). But, the compiler does not produce a default constructor if we write default constructor or parameterized constructor.

Example-10:

```
class Rectangle
{
     int length, width;
     Rectangle()      // Default constructor
     {
         length = 10;
         width  = 20;
     }
}
```

Object creation statement for the default constructor as below:

```
Rectangle rect1 = new Rectangle();      // default constructor
```

2. Parameterized constructor

Parameterized constructor is a constructor that takes parameters. If you want to initialize fields of the class with your own values, then use a parameterized constructor.

Example-11:

```
class Rectangle
{
     int length, width;
     Rectangle(int x, int y)       // Parameterized constructor
     {
         length = x;
         width = y;
     }
}
```

Object creation statement for the parameterized constructor as below:

```
Rectangle rect1 = new Rectangle(10, 20);   // parameterized constructor
```

- **Overloading Constructors**

Sometimes there is a need of initializing an object in different ways. Java allows to use more than one constructor in the same class. This can be done using constructor overloading. Constructor overloading is a technique in which a class can have more than one constructor that differ in parameter list, in such a way so that each constructor initializes an object in different ways.

Example-12:

Let us consider the **Rectangle** class with overloaded constructors:

```
class Rectangle
{
   int length, width;
   Rectangle()
   {
         length = 25;
         width  = 40;
   }
   Rectangle(int x, int y)
   {
         length = x;
         width = y;
   }
}
```

Here two constructors are used for class `Rectangle`. One is the default constructor and another is parameterized constructor. Following statements creates objects of class:

```
Rectangle rect1 = new Rectangle();             // default constructor
Rectangle rect2 = new Rectangle(15, 10);       // parameterized constructor
```

The `rect1` object is created and initialized the instance variables `length` and `width` to the default values `25` and `40` respectively. The `rect2` object is created and initialized the instance variables `length` and `width` to `15` and `10` respectively.

this Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the ***this*** keyword. The ***this*** keyword can be used inside any method to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked. Thus the use of **this** anywhere a reference to an object of the current class.

Example-13:

```
Box(double w, double h, double d)
{
        this.width = w;
        this.height = h;
        this.depth = d;
}
```

Inside the **Box( )**, **this** will always refer to the invoking object. The **this** is also useful in other contexts, one of which is explained in the following section.

Instance Variable Hiding

When a local variable has the same name as an instance variable, the local variable *hides* the instance variable. This is why in example-13, **width**, **height**, and **depth** were not used as the names of the parameters to the **Box( )** constructor inside the **Box** class. If they had been, then **width** would have referred to the formal parameter, hiding the instance variable **width**. Because **this** lets you refer directly to the object, you can use it to resolve any name space collisions that might occur between instance variables and local variables. For example, the local variables have names **width**, **height**, and **depth** for parameter names and then use **this** to access the instance variables by the same name.

Example-14:

```
Box(double width, double height, double depth)
{
        this.width = width;
        this.height = height;
        this.depth = depth;
}
```

Polymorphism

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations.

- ***Types of Java Polymorphism***

  In Java Polymorphism is mainly divided into two types:

  1. **Compile-time Polymorphism:** It is also known as static polymorphism. Furthermore, the call to the method is resolved at compile-time. This type of polymorphism is achieved by *Method Overloading*.
  2. **Runtime Polymorphism:** It is also known as Dynamic Binding or Dynamic Method Dispatch. In this, the call to an overridden method is resolved dynamically at runtime rather than at compile-time. This is achieved by *Method Overriding*.

Overloading Method

In Java it is possible to define two or more methods within the same class that share the same name, but different parameter list and different definitions. Each parameter list should be unique. Such methods are said to be *overloaded*, and the process is referred to as ***method overloading***. Method overloading is used, when objects are required to perform similar tasks but using different input parameters. Method overloading is one of the ways that Java supports polymorphism. Note that, the method's return type does not play any role in method overloading.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Example-15:

Here is a simple example that illustrates method overloading.

```
// Demonstrate method overloading.
class OverloadDemo
{
      void test()
      {
            System.out.println("No parameters");
      }
      // Overload test for one integer parameter.
      void test(int a)
      {
            System.out.println("a: " + a);
      }
      // Overload test for two integer parameters.
      void test(int a, int b)
      {
            System.out.println("a and b: " + a + " " + b);
      }
      // overload test for a double parameter
      double test(double a)
      {
            System.out.println("double a: " + a);
            return a*a;
      }
}
class Overload
{
      public static void main(String args[])
      {
            OverloadDemo ob = new OverloadDemo();
            double result;
            // call all versions of test()
            ob.test();
            ob.test(10);
            ob.test(10, 20);
            result = ob.test(123.25);
            System.out.println("Result of ob.test(123.25): " + result);
      }
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

In above example, **test( )** is overloaded four times. The first version takes no parameters, the second takes one integer parameter, and the third takes two integer parameters and the fourth takes one **double** parameter and also returns a value.

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters. However, this match need not always be exact. In some cases, Java's automatic type conversions can play a role in overload resolution.

Inheritance

In Java, it is possible to inherit attributes and methods from one class to another. This is done using the concept called *inheritance*.

Inheritance is a mechanism in which one class acquires the properties (instance-variables and methods) of another class. Inheritance is a mechanism of deriving new class from an old class. The old class is known as the *super class* or *base class* or *parent class* and the new one is called the *subclass* or *derived class* or *child class*. The inheritance allows subclass to inherit all the variables and methods of their parent classes.

With inheritance, we can reuse the fields and methods of the existing class. Hence, inheritance facilitates ***Reusability***. So that a class has to write only the unique features and rest of the common properties can be extended from another class.

In the terminology of Java, a class that is inherited is called a ***superclass***. The class that does the inheriting is called a ***subclass***. Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass.

In Java, to inherit from a class, use the **extends** keyword. Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

A subclass is defined as follows:

```
class subclassname extends superclassname
{
        Instance-variables declaration;
        methods declaration;
}
```

The keyword **extends** signifies that the properties of the *superclassname* are extended to *subclassname*. The subclass will now contain its own variables and methods as well those that of superclass.

Different types of inheritance in Java

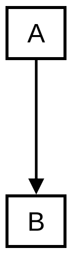
Inheritance may take different forms:

1) Single inheritance (only one super class)

2) Multilevel inheritance (Derived from a derived class)
3) Hierarchical inheritance (one super class, many subclasses)
4) Multiple inheritance (several super classes)

Java does not directly implement multiple inheritance. However, this concept is implemented using a secondary inheritance path in the form of ***interfaces***.

1) Single Inheritance



[Fig.: *Single Inheritance*]

When one subclass inherits the features of only one superclass, this type of inheritance is called Single inheritance. In the example given below, the base class Room will be inherited by a subclass BedRoom.

Example-16:

```
class Room
{
      int length;
      int breadth;
      Room(int x, int y)
      {
            length = x;
            breadth = y;
      }
      int area()
      {
            return (length * breadth);
      }
}
class BedRoom extends Room              // Inheriting Room
{
      int height;
      BedRoom(int x, int y, int z)
      {
            super(x, y);                //pass values to superclass
            height = z;
      }
      int volume()
      {
            return (length * breadth * height);
      }
}
```

Above program segment defines a class **`Room`** and extends it to another class **`BedRoom`**. Note that the class **`BedRoom`** defines its own data members and methods. The subclass **`BedRoom`** now includes three instance variables, namely, **`length`**, **`breadth`**, and **`height`** and two methods, **`area`** and **`volume`**.

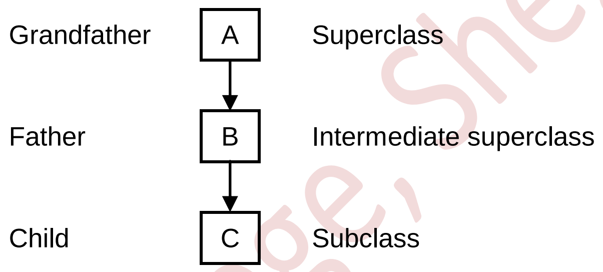
Subclass Constructor

A subclass constructor is used to construct the instance variables of both the subclass and the superclass. The subclass constructor uses the keyword **`super`** to invoke the constructor method of the superclass. The keyword **`super`** is used subject to the following conditions:

- **`super`** may only be used within a subclass constructor method.
- The call to superclass constructor must appear as the first statement within the subclass constructor
- The parameters in the **`super`** call must match the order and type of the instance variable declared in the superclass.

2) Multilevel Inheritance

A common requirement in object-oriented programming is the use of a derived class as a super class, Java supports this concept through the use of multilevel inheritance and uses it extensively in building its class library. This concept allows us to build a chain of classes as shown in Fig.



[Fig,: *Multilevel Inheritance*]

The class A serves as a base class for the derived class B which in turn serves as a base class for the derived class C. The chain ABC is known as *inheritance path*.

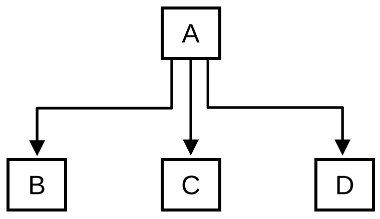
A derived class with multilevel base classes is declared as follows:

```
class A
{
        .............
        .............
}
class B extends A              // First Level
{
        .............
        .............
}
class C extends B              // Second Level
{
        .............
        .............
}
```

This process may be extended to any number of levels. The class C can inherit the members of both A and B.

3) Hierarchical Inheritance

One of the applications of inheritance is to use it as a support to the hierarchical design of a program. In the case of Hierarchical Inheritance, there is one base class for multiple subclasses. For the example given below, A is the base class that is inherited by multiple subclasses B, C, and D.



[Fig.: *Hierarchical Inheritance*]

Overriding Methods

If subclass (child class) has the same method as declared in the parent class, it is known as ***method overriding*** in Java. That is a method in a subclass has the same name and type signature as a method in its superclass. In other words, if a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

The main uses of method overriding are to provide the specific implementation of a method which is already provided by its superclass and also used for runtime polymorphism.

Example-17:

Consider the following program segment that illustrates the concept of overriding. The method `getROI()` is overridden.

```
class Bank
{
      int getROI()          // Method declared
      {
         return 0;
      }
}
class SBI extends Bank
{
     int getROI()           // Method Override
     {
         return 8;
     }
}
class AXIS extends Bank
{
     int getROI()           // Method Override
     {
         return 9;
     }
}
```

Abstract class

Data abstraction is the process of hiding certain details and showing only essential information to the user. Abstraction can be achieved with *abstract classes*. By using abstract classes, a foundation is established for other classes to build upon, promoting code reusability, data abstraction and inheritance.

A class which is declared with the `abstract` keyword is known as ***Abstract class***. A abstract class serves as a blueprint for other classes and can contain abstract methods (methods without a body) and non-abstract methods (methods with a body). Abstract classes in Java cannot be directly instantiated, which means you cannot create objects of an abstract class. It is expected to be extended by other classes implementing its abstract methods. A abstract class needs to be subclassed by another class to use its properties.

Abstract methods declared in a superclass are compulsorily override or use in subclasses (otherwise you will get compilation error). The body is provided by the subclass (inherited from). To declare an abstract method, use this general form:

```
abstract type name(parameter-list);
```

Example:

```
abstract class Shape
{
      abstract void draw();
}

class Rectangle extends Shape
{
      void draw()
      {
            System.out.println("drawing rectangle");
      }
}
class Circle1 extends Shape
{
      void draw()
      {
            System.out.println("drawing circle");
      }
}
```

Interfaces

In Java, an interface specifies the behavior of a class by providing an abstract type. It is one of the core concepts in Java and is used to achieve abstraction, polymorphism and multiple inheritances.

Interface looks like a class but it is not a class. A Java interface can contains only abstract methods (method without body) and static constant variables. In other words, interface fields are public, static and final by default, and the methods are public and abstract. An Interface cannot contain a constructor methods and also it cannot be instantiated.

Java class cannot be a subclass of more than one superclass, but java class can implement more than one interface, thus support the concept of *multiple inheritance* through interfaces. Once interface is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.

- ***Defining Interface***

The syntax for defining an interface is very similar to that for defining a class:

```
interface interfaceName
{
     variables declaration;
     methods declaration;
}
```

Here, *`interface`* is the key word and *`interfaceName`* is any valid Java identifier.

Note that, all variables are declared as constants. Methods declaration will contain only a list of methods without a body statements as shown below:

```
return-type methodName(parameter_list);
```

Example:-

```
interface MotorBike
{
      int speed=50;
      public void totalDistance();
}
```

Note that the code for the method is not included in the interface and the class that implements this interface must define the code for the method.

- ***Implementing Interfaces***

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the `implements` keyword (instead of `extends` keyword) in a class definition, and then create the methods defined by the interface. That is, the class which implements the interface needs to provide complete body for the methods declared in the interface. The general form of a class that includes the `implements` clause looks like this:

```
class classname implements interface[,interface,...]
{
    Body of class
}
```

If a class implements more than one interface, the interfaces are separated with a comma. The methods that implement an interface must be declared `public`. Also, the type signature of the implementing method must match exactly the type signature specified in the `interface` definition. To implement an interface, a class must create the complete set of methods defined by the interface.

Example:

```
public class TwoWheeler implements MotorBike
{
   int totalDistance;
   public void totalDistance()
   {
      totalDistance = speed*150;
      System.out.println("Distance Travelled:" + totalDistance);
   }
}
```

Using Final (variables, methods and classes)

All methods and variables can be overridden by default in subclasses. If we wish to prevent the subclasses from overriding the members of the superclass, we can declare them as final using the keyword **final** as a modifier.

In java **final** is a keyword or reserved word and can be applied to variables, methods, classes etc. The reason behind **final** keyword is to make entity non modifiable. It means when you make a variable or class or method as final you are not allowed to change that variable or class or method.

The **final** keyword in java is used to give restriction to the user. The java final keyword can be used in many contexts, as:

• final variable	→	To create constant variables
• final method	→	Prevent Method overriding
• final class	→	Prevent Inheritance

Final variables

A variable can be declared as final. Any variable which is declared by using the **final** keyword is called ***final variable***. Final variables are treated as constant and prevent its contents from being modified. Final variables must be initialized when it is declared. This means that a final variable can only be explicitly assigned once. Example:

```
final int SIZE = 100;
final int hours = 24;
```

Blank final variable

A final variable that is not initialized at the time of declaration is known as **blank final variable**. The blank final variable must be initialized in constructor of the class otherwise it will throw a compilation error. Example:

```
class Demo
{
      // Blank final variable
      final int MAX_VALUE;
      Demo()
      {
            //It must be initialized in constructor
            MAX_VALUE=100;
      }
}
```

Final methods

Method overriding is one of Java's most powerful features, but sometimes it is required to prevent a *subclass* to overriding a method from *superclass*. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration in a superclass. A method with final keyword is called ***final method***. Final methods cannot be overridden. A sub class can call the final method of super class, but it cannot override it.

The final methods are faster than non-final methods because they are not required to be resolved during run-time and they are bonded on compile time. The main reason behind making a method final would be that the content of the method should not be changed by any outsider. Any attempt to override the final method will result in compilation error.

Example :

```
class Base
{
     final void display()
     {
          System.out.println("Base method called");
     }
}
class Derived extends Base
{
     void display() //cannot override
     {
          System.out.println("Base method called");
     }
}
```

Final classes

When a class is declared with **final** keyword, it is known as ***final class*** in java. A final class cannot be extended (inherited). The main purpose or reason of using a final class is to prevent inheritance as final classes cannot be extended. If a class is marked as final then no class can inherit any feature from the final class.

Example :-

```
final class X
{
      //properties and methods of class X
}
class Y extends X
{
      //properties and methods of class Y
}
```

Any attempt to inherit these classes will cause an error and the compiler will not allow it.

```
Compiler Error: cannot inherit from final X
```

Garbage Collection

In java, garbage means unreferenced objects. Garbage collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects. Unlike C and C++, in java garbage collection is performed automatically. So, java provides better memory management.

Garbage collection in Java is an automatic memory management process that frees up memory occupied by objects that are no longer referenced or in use by the program. In Java, developers don't explicitly deallocate memory; instead, the Java Virtual Machine (JVM) handles this task through its garbage collection mechanism.

The overview of garbage collection works in Java is as follows:

1. **Allocation:** When you create objects in Java using the '`new`' keyword, memory is allocated for those objects on the heap.
2. **Reference tracking:** Java keeps track of references to objects. An object becomes eligible for garbage collection when it is no longer reachable by any live thread or chain of references.

3. **Mark and Sweep:** The most common garbage collection algorithm used in Java is the Mark and Sweep algorithm. Here's how it works:
   a) **Mark phase:** The garbage collector traverses all reachable objects starting from the roots (which typically include local variables, static variables, and active threads) and marks them as live.
   b) **Sweep phase:** The garbage collector sweeps through the heap and deallocates memory for objects that were not marked as live, effectively reclaiming that memory.
4. **Compact:** Some garbage collectors in Java also perform memory compaction after reclaiming memory. Compaction involves moving live objects together to eliminate fragmented memory spaces, which can improve memory usage and performance.

Java provides different garbage collection implementations, including:

- Serial Garbage Collector
- Parallel Garbage Collector
- Concurrent Mark-Sweep (CMS) Garbage Collector
- Garbage-First (G1) Garbage Collector

The choice of garbage collector depends on factors such as application requirements, available memory, and performance considerations. In recent versions of Java, the Garbage-First (G1) garbage collector has become the default for most applications due to its ability to provide better overall performance and scalability.

Developers can also tune garbage collection behavior by adjusting JVM options such as heap size, garbage collection algorithm, and collection frequency.

Overall, garbage collection in Java simplifies memory management for developers by automating memory deallocation, reducing the risk of memory leaks and segmentation faults, and allowing them to focus more on application logic.

■ ■ ■ ■ ■

References:

1) The Complete Reference JAVA2 by Herbert Schildt (Tata McGraw)
2) The Complete Reference JAVA by Patrik Noughton
3) Programming with JAVA - A Primer : By E. Balguruswamy (Tata McGraw)
4) Programming in JAVA : By S. S. Khandare (S. Chand)
5) Teach Yourself Java in 2 Hrs : By Sams.
6) Java for You : By P. Koparkar

■ ■ ■ ■ ■