MODULE II
CLASSES

STRUCTURE

LEARNING OBJECTIVES

- To understand the Class fundamentals.
- To learn how to declare and assign object reference variables.
- To understand the concept of Inheritance.

2.1 INTRODUCTION

Java is an object-oriented programming language. Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.A Class is like an object constructor, or a "blueprint" for creating objects.

2.2 CLASS FUNDAMENTALS, DECLARING OBJECTS, ASSIGNING OBJECT REFERENCE VARIABLES

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

Modifiers : A class can be public or has default access
Class name: The name should begin with a initial letter (capitalized by convention).
Superclass(if any): The name of the class’s parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
Interfaces(if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
Body: The class body surrounded by braces, { }.

Create a Class
To create a class, use the keyword class:
MyClass.java
Create a class named "MyClass" with a variable x:

```
public class MyClass {
  int x = 5;
```

Create an Object
In Java, an object is created from a class. We have already created the class named MyClass, so now we can use this to create objects.

To create an object of MyClass, specify the class name, followed by the object name, and use the keyword new:

Example
Create an object called "myObj" and print the value of x:

```
public class MyClass {
  int x = 5;
  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    System.out.println(myObj.x);
  }
}
```

Multiple Objects
You can create multiple objects of one class:
Example
Create two objects of MyClass:

```
public class MyClass {
  int x = 5;
  public static void main(String[] args) {
    MyClass myObj1 = new MyClass();  // Object 1
    MyClass myObj2 = new MyClass();  // Object 2
    System.out.println(myObj1.x);
    System.out.println(myObj2.x);
  }
}
```

Using Multiple Classes
You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the main() method (code to be executed)).Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory/folder:
MyClass.java
OtherClass.java
MyClass.java

```
public class MyClass {
  int x = 5;
}
```

OtherClass.java

```
class OtherClass {
  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    System.out.println(myObj.x);
  }
}
```

When both files have been compiled:

```
C:\Users\Your Name>javac MyClass.java
```

```
C:\Users\Your Name>javac OtherClass.java
```
Run the OtherClass.java file:
```
C:\Users\Your Name>java OtherClass
```
And the output will be:
```
5
```

2.3 METHODS, CONSTRUCTORS, "THIS" KEYWORD, FINALIZE ( ) METHOD A STACK CLASS, OVER LOADING METHODS

final(lowercase) is a reserved keyword in java. We can't use it as an identifier as it is reserved. We can use this keyword with variables, methods and also with classes. The final keyword in java has different meaning depending upon it is applied to variable, class or method.

1. **final with Variables :** The value of variable cannot be changed once initialized.
   ```
   class A {
      public static void main(String[] args)
      {
         // Non final variable
         int a = 5;

         // final variable
         final int b = 6;

         // modifying the non final variable : Allowed
         a++;

         // modifying the final variable :
         // Immediately gives Compile Time error.
         b++;
      }
   }
   ```
   If we declare any variable as final, we can't modify its contents since it is final, and if we modify it then we get Compile Time Error.

2. **final with Class :** The class cannot be subclasses. Whenever we declare any class as final, it means that we can't extend that class or that class can't be extended or we can't make subclass of that class.
   ```
   final class RR {
      public static void main(String[] args)
      {
         int a = 10;
      }
   }
   // here gets Compile time error that
   // we can't extend RR as it is final.
   class KK extends RR {
      // more code here with main method
   }
   ```

final with Method : The method cannot be overridden by a subclass. Whenever we declare any method as final, then it means that we can't override that method.

filter_none
brightness_4

```
class QQ {
   final void rr() {}
   public static void main(String[] args)
   {
   }
}

class MM extends QQ {

   // Here we get compile time error
   // since can't extend rr since it is final.
   void rr() {}
}
```

Note : If a class is declared as final then by default all of the methods present in that class are automatically final but variables are not.

```
/ Java program to illustrate final keyword
final class G {

   // by default it is final.
   void h() {}

   // by default it is not final.
   static int j = 30;

public static void main(String[] args)
   {
      // See modified contents of variable j.
      j = 36;
      System.out.println(j);
   }
}
```

Output:

```
36
```

finally keyword

Just as final is a reserved keyword, so in same way finally is also a reserved keyword in java i.e, we can’t use it as an identifier. The finally keyword is used in association with a try/catch block and guarantees that a section of code will be executed, even if an exception is thrown. The finally block will be executed after the try and catch blocks, but before control transfers back to its origin.

```
// A Java program to demonstrate finally.
class Geek {
   // A method that throws an exception and has finally.
   // This method will be called inside try-catch.
   static void A()
   {
      try {
         System.out.println("inside A");
         throw new RuntimeException("demo");
```

```
        }
        finally
        {
            System.out.println("A's finally");
        }
    }

    // This method also calls finally. This method
    // will be called outside try-catch.
    static void B()
    {
        try {
            System.out.println("inside B");
            return;
        }
        finally
        {
            System.out.println("B's finally");
        }
    }

    public static void main(String args[])
    {
        try {
            A();
        }
        catch (Exception e) {
            System.out.println("Exception caught");
        }
        B();
    }
}
```

Output:

```
inside A
A's finally
Exception caught
inside B
B's finally
```

There are various cases when finally can be used. There are discussed below:

Case 1 : Exceptions do not occur in the program

```
// Java program to illustrate finally in
// Case where exceptions do not
// occur in the program
class B {
    public static void main(String[] args)
    {
        int k = 55;
        try {
            System.out.println("In try block");
            int z = k / 55;
        }

        catch (ArithmeticException e) {
            System.out.println("In catch block");
            System.out.println("Dividing by zero but caught");
        }

        finally
        {
            System.out.println("Executes whether exception occurs or not");
        }
    }
}
```

Output:

```
In try block
Executes whether exception occurs or not
```

Here above exception not occurs but still finally block executes since finally is meant to execute whether an exception occurs or not.

Flow of Above Program: First it starts from the main method and then goes to try block and in try since no exception occurs so flow doesn’t goes to catch block hence flow goes directly from try to finally block.

Case 2 : Exception occurs and corresponding catch block matches

```
// Java program to illustrate finally in
// Case where exceptions occur
// and match in the program
class C {
    public static void main(String[] args)
    {
        int k = 66;
        try {
            System.out.println("In try block");
            int z = k / 0;
            // Carefully see flow dosen't come here
            System.out.println("Flow dosen't came here");
        }

        catch (ArithmeticException e) {
            System.out.println("In catch block");
            System.out.println("Dividing by zero but caught");
        }

        finally
        {
            System.out.println("Executes whether an exception occurs or not");
        }
    }
}
```

Output:

```
In try block
In catch block
Dividing by zero but caught
Executes whether an exception occurs or not
```

Here, the above exception occurs and corresponding catch block found but still finally block executes since finally is meant to execute whether an exception occurs or not or whether corresponding catch block found or not.

Flow of Above Program: First, starts from the main method and then goes to try block and in try an Arithmetic exception occurs and the corresponding catch block is also available so flow goes to catch block. After that flow doesn’t go to try block again since once an exception occurs in try block then flow dosen’t come back again to try block. After that finally, execute since finally is meant to execute whether an exception occurs or not or whether corresponding catch block found or not.

Case 3 : Exception occurs and corresponding catch block not found/match

brightness_4

```
// Java program to illustrate finally in
// Case where exceptions occur
// and do not match any case in the program
class D {
   public static void main(String[] args)
   {
      int k = 15;
      try {
         System.out.println("In try block");
         int z = k / 0;
      }
      catch (NullPointerException e) {
         System.out.println("In catch block");
         System.out.println("Dividing by zero but caught");
      }

      finally
      {
         System.out.println("Executes whether an exception occurs or not");
      }
   }
}
```

Output:

```
In try block
Executes whether an exception occurs or not
Exception in thread "main":java.lang.ArithmeticException:
/ by zero followed by stack trace.
```

Here above exception occurs and corresponding catch block not found/match but still finally block executes since finally is meant to execute whether an exception occurs or not or whether corresponding catch block found/match or not.

Flow of Above Program: First starts from main method and then goes to try block and in try an Arithmetic exception occurs and corresponding catch block is not available so flow dosen't goes to catch block. After that flow doesn't go to try block again since once an exception occurs in try block then flow dosen't come back again to try block. After that finally, execute since finally is meant to execute whether an exception occurs or not or whether corresponding catch block found/match or not.

Application of finally block: So basically the use of finally block is resource deallocation. Means all the resources such as Network Connections, Database Connections, which we opened in try block are needed to be closed so that we won't lose our resources as opened. So those resources are needed to be closed in finally block.
finalize method

It is a method that the Garbage Collector always calls just before the deletion/destroying the object which is eligible for Garbage Collection, so as to perform clean-up activity. Clean-up activity means closing the resources associated with that object like Database Connection, Network Connection or we can say resource de-allocation. Remember it is not a reserved keyword.
Once the finalize method completes immediately Garbage Collector destroy that object. finalize method is present in Object class and its syntax is:

protected void finalize throws Throwable{}
Since Object class contains the finalize method hence finalize method is available for every java class since Object is the superclass of all java classes. Since it is available for every java class hence Garbage Collector can call finalize method on any java object
Now, the finalize method which is present in the Object class, has an empty implementation, in our class clean-up activities are there, then we have to override this method to define our own clean-up activities.

Cases related to finalize method:
Case 1: The object which is eligible for Garbage Collection, that object's corresponding class finalize method is going to be executed
brightness_4

```
class Hello {
   public static void main(String[] args)
   {
      String s = new String("RR");
      s = null;

      // Requesting JVM to call Garbage Collector method
      System.gc();
      System.out.println("Main Completes");
   }

   // Here overriding finalize method
   public void finalize()
   {
      System.out.println("finalize method overriden");
   }
}
```

Output:

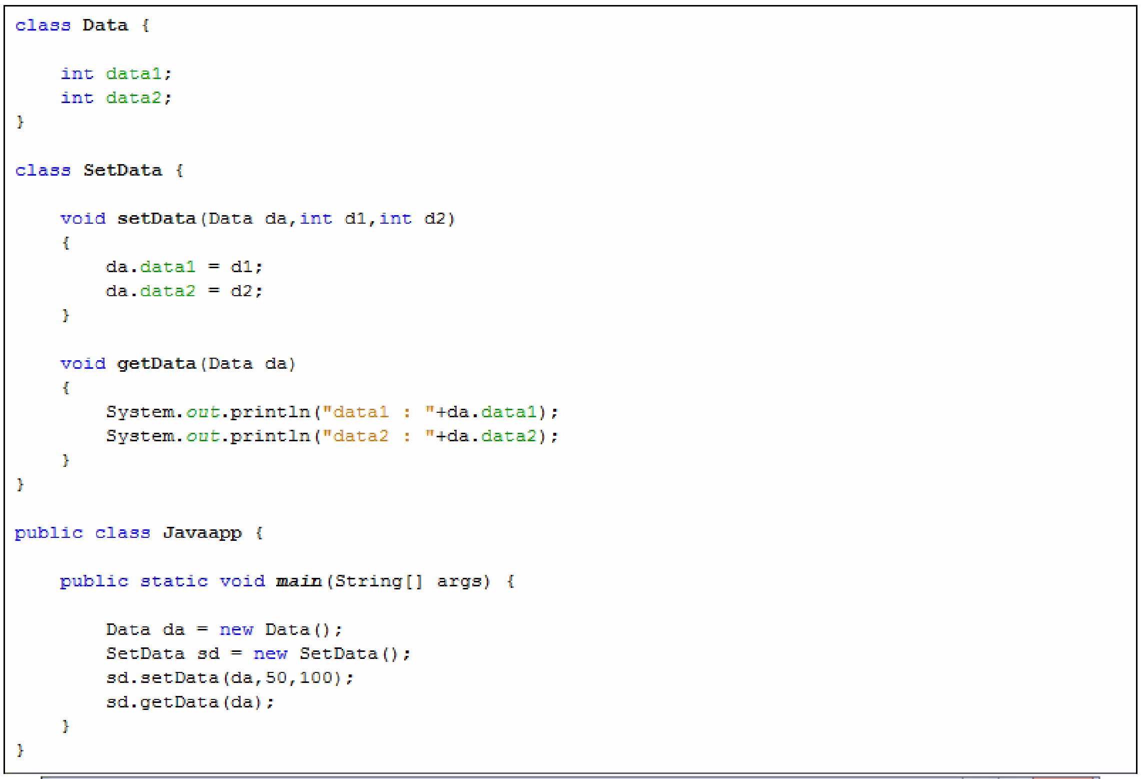
```
Main Completes
```

Note : Here above output came only Main Completes and not “finalize method overriden” because Garbage Collector call finalize method on that class object which is eligible for Garbage collection. Here above we have done->
s = null and ‘s’ is the object of String class, so String class finalize method is going to be called and not our class(i.e, Hello class). So we modify our code like->

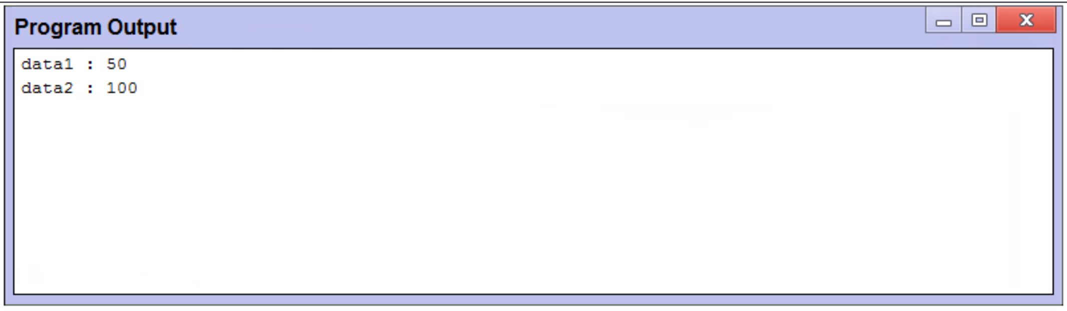
2.4 USING OBJECTS AS PARAMETERS, ARGUMENT PASSING, RETURNING OBJECTS, RECURSION, ACCESS CONTROL

Objects as Parameters : A method can take an objects as a parameter. For example, in the following program, the method setData( ) takes three parameter. The first parameter is an Data object. If you pass an object as an argument to a method, the mechanism that applies is called pass-by-reference, because a copy of the reference contained in the variable is transferred to the method, not a copy of the object itself.

Program



```
class Data {

    int data1;
    int data2;
}

class SetData {

    void setData(Data da,int d1,int d2)
    {
        da.data1 = d1;
        da.data2 = d2;
    }

    void getData(Data da)
    {
        System.out.println("data1 : "+da.data1);
        System.out.println("data2 : "+da.data2);
    }
}

public class Javaapp {

    public static void main(String[] args) {

        Data da = new Data();
        SetData sd = new SetData();
        sd.setData(da,50,100);
        sd.getData(da);
    }
}
```



```
data1 : 50
data2 : 100
```

Program Source
```
class Data {

   int data1;
   int data2;
}

class SetData {

   void setData(Data da,int d1,int d2)
   {
      da.data1 = d1;
      da.data2 = d2;
   }

   void getData(Data da)
   {
      System.out.println("data1 : "+da.data1);
      System.out.println("data2 : "+da.data2);
   }
}

public class Javaapp {

   public static void main(String[] args) {

      Data da = new Data();
      SetData sd = new SetData();
      sd.setData(da,50,100);
      sd.getData(da);
   }
}
```
Argument passing- In Java methods, arguments are passed by value. When invoked, the method receives the value of the variable passed in. When the argument is of primitive type, pass-by-value means that the method cannot change its value. When the argument is of reference type, pass-by-value means that the method cannot change the object reference, but can invoke the object's methods and modify the accessible variables within the object.
This is often the source of confusion--a programmer writes a method that attempts to modify the value of one its arguments and the method doesn't work as expected. Let's look at such method and then investigate how to change it so that it does what the programmer originally intended.

Consider this series of Java statements which attempts to retrieve the current color of a Pen object in a graphics application:

```
. . .
int r = -1, g = -1, b = -1;
pen.getRGBColor(r, g, b);
System.out.println("red = " + r + ", green = " + g + ", blue = " + b);
. . .
```

At the time when the getRGBColor method is called, the variables r, g, and b all have the value -1. The caller is expecting the getRGBColor method to pass back the red, green and blue values of the current color in the r, g, and b variables.
However, the Java runtime passes the variables' values (-1) into the getRGBColor method; not a reference to the r, g, and b variables. So you could visualize the call to getRGBColor like this: getRGBColor(-1, -1, -1).

When control passes into the getRGBColor method, the arguments come into scope (get allocated) and are initialized to the value passed into the method:

```
class Pen {
   int redValue, greenValue, blueValue;
   void getRGBColor(int red, int green, int blue) {
      // red, green, and blue have been created
      // and their values are -1
      . . .
   }
}
```

So getRGBColor gets access to the values of r, g, and b in the caller through its arguments red, green, and blue, respectively. The method gets its own copy of the values to use within the scope of the method. Any changes made to those local copies are not reflected in the original variables from the caller.
Now, let's look at the implementation of getRGBColor within the Pen class that the method signature above implies:

```
class Pen {
   int redValue, greenValue, blueValue;
   . . .
      // this method does not work as intended
   void getRGBColor(int red, int green, int blue) {
      red = redValue;
      green = greenValue;
      blue = blueValue;
   }
}
```

This method will not work as intended. When control gets to the println statement in the following code, which was shown previously, getRGBColor's arguments, red, green, and blue, no longer exist. Therefore the assignments made to them within the method had no effect; r, g, and b are all still equal to -1.

```
. . .
int r = -1, g = -1, b = -1;
pen.getRGBColor(r, g, b);
System.out.println("red = " + r + ", green = " + g + ", blue = " + b);
. . .
```

Passing variables by value affords the programmer some safety: Methods cannot unintentionally modify a variable that is outside of its scope. However, you often want a method to be able to modify one or more of its arguments. The getRGBColor method is a case in point. The caller wants the method to return three values through its arguments. However, the method cannot modify its arguments, and, furthermore, a method can only return one value through its return value. So, how can a method return more than one value, or have an effect (modify some value) outside of its scope?

For a method to modify an argument, it must be of a reference type such as an object or array. Objects and arrays are also passed by value, but the value of an object is a reference. So the effect is that arguments of reference types are passed in by reference. Hence the name. A reference to an object is the address of the object in memory. Now, the argument in the method is referring to the same memory location as the caller.

Let's rewrite the getRGBColor method so that it actually does what you want. First, you must introduce a new type of object, RGBColor, that can hold the red, green and blue values of a color in RGB space:

```
class RGBColor {
   public int red, green, blue;
}
```

Now, we can rewrite getRGBColor so that it accepts an RGBColor object as an argument. The getRGBColor method returns the current color of the pen by setting the red, green and blue member variables of its RGBColor argument:

```
class Pen {
   int redValue, greenValue, blueValue;
   void getRGBColor(RGBColor aColor) {
      aColor.red = redValue;
      aColor.green = greenValue;
      aColor.blue = blueValue;
   }
}
```

And finally, let's rewrite the calling sequence:

```
. . .
RGBColor penColor = new RGBColor();
pen.getRGBColor(penColor);
System.out.println("red = " + penColor.red + ", green = " +
                        penColor.green + ", blue = " + penColor.blue);
. . .
```

The modifications made to the RGBColor object within the getRGBColor method affect the object created in the calling sequence because the names penColor (in the calling sequence) and aColor (in the getRGBColor method) refer to the same object.

Recursion- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc. In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

```
int fact(int n)
{
   if (n < = 1) // base case
      return 1;
   else
      return n*fact(n-1);
}
```

In the above example, base case for n < = 1 is defined and larger value of number can be solved by converting to smaller one till base case is reached.

Access control- We can control the access level for class member variables and methods through access specifiers.
Java's access specifiers are public, private, protected and a default access level.

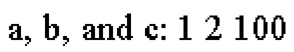
Level

- A public class member can be accessed by any other code.
- A private class member can only be accessed within its class.
- A default access class member has no access specifiers. A class's default features are accessible to any class in the same package.
- A protected feature of a class is available to all classes in the same package(like a default) and to its subclasses.
- protected features are more accessible than default features.

Example
To understand the effects of public and private access, consider the following program:

```
class Test {
  int a;        // default access
  public int b; // public access
  private int c; // private access
  // methods to access c
  void setc(int i) {
    c = i;
  }
  int getc() {
    return c;
  }
}
public class Main {
  public static void main(String args[]) {
    Test ob = new Test();
    ob.a = 1;
    ob.b = 2;
    // This is not OK and will cause an error
    // ob.c = 100; // Error!
    // You must access c through its methods
    ob.setc(100); // OK
    System.out.println("a, b, and c: " + ob.a +
         " " + ob.b + " " + ob.getc());
  }
}
```

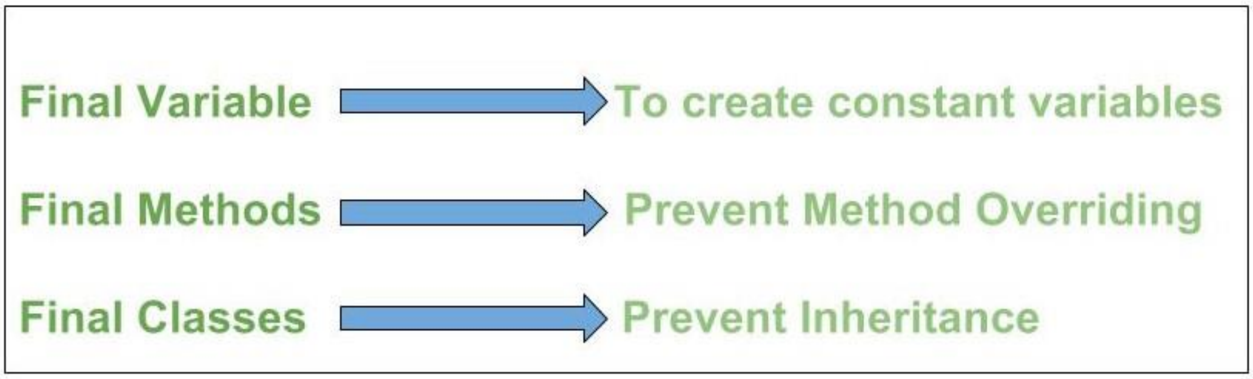
The output:



```
a, b, and c: 1 2 100
```

2.5 INTRODUCING FINAL, UNDERSTANDING STATIC, INTRODUCING NESTED AND INNER CLASSES, USING COMMAND LINE ARGUMENTS

final keyword is used in different contexts. First of all, *final* is a non-access modifier applicable only to a variable, a method or a class.Following are different contexts where final is used.



When a variable is declared with *final* keyword, its value can't be modified, essentially, a constant. This also means that you must initialize a final variable. If the final variable is a reference, this means that the variable cannot be re-bound to reference another object, but internal state of the object pointed by that reference variable can be changed i.e. you can add or remove elements from final array or final collection. It is good practice to represent final variables in all uppercase, using underscore to separate words.

Examples :

```
// a final variable
final int THRESHOLD = 5;
// a blank final variable
final int THRESHOLD;
// a final static variable PI
static final double PI = 3.141592653589793;
// a  blank final static  variable
static final double PI;
```

1. **Initializing a final variable :**
   We must initialize a final variable, otherwise compiler will throw compile-time error.A final variable can only be initialized once, either via an initializer or an assignment statement.
   There are three ways to initialize a final variable :
2. You can initialize a final variable when it is declared.This approach is the most common. A final variable is called **blank final variable**,if it is **not** initialized while declaration. Below are the two ways to initialize a blank final variable.
3. A blank final variable can be initialized inside instance-initializer block or inside constructor. If you have more than one constructor in your class then it must be initialized in all of them, otherwise compile time error will be thrown.
4. A blank final static variable can be initialized inside static block.
   Let us see above different ways of initializing a final variable through an example.

```
//Java program to demonstrate different
// ways of initializing a final variable

class Gfg
{
   // a final variable
   // direct initialize
   final int THRESHOLD = 5;
```

```
    // a blank final variable
    final int CAPACITY;

    // another blank final variable
    final int  MINIMUM;

    // a final static variable PI
    // direct initialize
    static final double PI = 3.141592653589793;

    // a  blank final static  variable
    static final double EULERCONSTANT;

    // instance initializer block for
    // initializing CAPACITY
    {
        CAPACITY = 25;
    }

    // static initializer block for
    // initializing EULERCONSTANT
    static{
        EULERCONSTANT = 2.3;
    }

    // constructor for initializing MINIMUM
    // Note that if there are more than one
    // constructor, you must initialize MINIMUM
    // in them also
    public GFG()
    {
        MINIMUM = -1;
    }

}
```

When to use a final variable:
The only difference between a normal variable and a final variable is that we can re-assign value to a normal variable but we cannot change the value of a final variable once assigned. Hence final variables must be used only for the values that we want to remain constant throughout the execution of program.

Reference final variable:
When a final variable is a reference to an object, then this final variable is called reference final variable. For example, a final StringBuffer variable looks like
final StringBuffer sb;

As you know that a final variable cannot be re-assign. But in case of a reference final variable, internal state of the object pointed by that reference variable can be changed. Note that this is not re-assigning. This property of *final* is called *non-transitivity*. To understand what is mean by internal state of the object, see below example :

```
// Java program to demonstrate
// reference final variable

class Gfg
{
   public static void main(String[] args)
   {
      // a final reference variable sb
      final StringBuilder sb = new StringBuilder("Geeks");

      System.out.println(sb);

      // changing internal state of object
      // reference by final reference variable sb
      sb.append("ForGeeks");

      System.out.println(sb);
   }
}
```

Output:

```
Geeks
GeeksForGeeks
```

The *non-transitivity* property also applies to arrays, because arrays are objects in java. Arrays with final keyword are also called final arrays.

Understanding static- *static* is a non-access modifier in Java which is applicable for the following:

1. blocks
2. variables
3. methods
4. nested classes

To create a static member(block,variable,method,nested class), precede its declaration with the keyword *static*. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. For example, in below java program, we are accessing static method *m1()* without creating any object of *Test* class.

```
// Java program to demonstrate that a static member
// can be accessed before instantiating a class
class Test
{
   // static method
   static void m1()
   {
      System.out.println("from m1");
   }

   public static void main(String[] args)
   {
        // calling m1 without creating
        // any object of class Test
         m1();
```

```
    }
}
```

Output:

```
from m1
```

Static blocks

If you need to do computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded. Consider the following java program demonstrating use of static blocks.

```
// Java program to demonstrate use of static blocks
class Test
{
    // static variable
    static int a = 10;
    static int b;

    // static block
    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }

    public static void main(String[] args)
    {
       System.out.println("from main");
       System.out.println("Value of a : "+a);
       System.out.println("Value of b : "+b);
    }
}
```

Output:

```
Static block initialized.
from main
Value of a : 10
Value of b : 40
```

For Detailed article on static blocks, see static blocks

Static variables

When a variable is declared as static, then a single copy of variable is created and shared among all objects at class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.

Important points for static variables :-

- We can create static variables at class-level only.
- static block and static variables are executed in order they are present in a program.

Below is the java program to demonstrate that static block and static variables are executed in order they are present in a program.

```
// java program to demonstrate execution
// of static blocks and variables
```

```
class Test
{
    // static variable
    static int a = m1();

    // static block
    static {
        System.out.println("Inside static block");
    }

    // static method
    static int m1() {
        System.out.println("from m1");
        return 20;
    }

    // static method(main !!)
    public static void main(String[] args)
    {
       System.out.println("Value of a : "+a);
       System.out.println("from main");
    }


}
```

Output:

```
from m1
Inside static block
Value of a : 20
from main
```

Static methods

When a method is declared with *static* keyword, it is known as static method. The most common example of a static method is *main( )* method.As discussed above, Any static member can be accessed before any objects of its class are created, and without reference to any object.Methods declared as static have several restrictions:

- They can only directly call other static methods.
- They can only directly access static data.
- They cannot refer to this or super in any way.

Below is the java program to demonstrate restrictions on static methods.

```
// java program to demonstrate restriction on static methods
class Test
{
    // static variable
    static int a = 10;

    // instance variable
    int b = 20;

    // static method
    static void m1()
    {
        a = 20;
        System.out.println("from m1");

         // Cannot make a static reference to the non-static field b
         b = 10; // compilation error

         // Cannot make a static reference to the
                 // non-static method m2() from the type Test
         m2();  // compilation error

         //  Cannot use super in a static context
         System.out.println(super.a); // compiler error
    }

    // instance method
    void m2()
    {
        System.out.println("from m2");
    }

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

Introducing Nested and Inner classes- In Java, you can define a class within another class. Such class is known as nested class.

```
class OuterClass {
    // ...
    class NestedClass {
        // ...
    }
}
```

There are two types of nested classes you can create in Java.

- Non-static nested class (inner class)
- Static nested class

Let's first look at non-static nested class

Non-Static Nested Class-Non-static nested class is a class within another class, where the class has access to members of the enclosing class (outer class). It is commonly known as inner class.
Since, inner class exists within the outer class (in order to instantiate an inner class, you must first instantiate the outer class).
Here’s example how you can declare Inner classes in Java.

Example 1: Inner class

```
class CPU {
    double price;
    class Processor{
        double cores;
        String manufacturer;
        double getCache(){
            return 4.3;
        }
    }
    protected class RAM{
        double memory;
        String manufacturer;
        double getClockSpeed(){
            return 5.5;
        }
    }
}
public class Main {
    public static void main(String[] args) {
        CPU cpu = new CPU();
        CPU.Processor processor = cpu.new Processor();
        CPU.RAM ram = cpu.new RAM();
        System.out.println("Processor Cache = " + processor.getCache());
        System.out.println("Ram Clock speed = " + ram.getClockSpeed());
    }
}
```

When you run above program, the output will be:

```
Processor Cache = 4.3
Ram Clock speed = 5.5
```

In above program, the class CPU encapsulates two inner classes i.e. Processor and RAM. Since, the class RAM is inner class you can declared it as protected.
In the Main class, the instance of CPU is created at first. And in order to create the instance of Processor, the . (dot) operator is used.
CPU.Processor processor = cpu.new Processor();

Accessing Members of Outer Class within Inner Class
Like we discussed, inner classes can access the members of the outer class. This is possible using Java this keyword.

Example 2: Accessing Members

```
public class Car {
    String carName;
    String carType;
    public Car(String name, String type) {
        this.carName = name;
        this.carType = type;
    }
    private String getCarName() {
        return this.carName;
    }
    class Engine {
        String engineType;
        void setEngine() {
// Accessing carType property of Car
            if(Car.this.carType.equals("4WD")){
// Invoking method getCarName() of Car
                if(Car.this.getCarName().equals("Crysler")) {
                    this.engineType = "Bigger";
                } else {
                    this.engineType = "Smaller";
                }
            }else{
                this.engineType = "Bigger";
            }
        }
        String getEngineType(){
            return this.engineType;
        }
    }
}
public class CarMain {
    public static void main(String[] args) {
        Car car1 = new Car("Mazda", "8WD");
        Car.Engine engine = car1.new Engine();
        engine.setEngine();
        System.out.println("Engine Type for 8WD= " + engine.getEngineType());

        Car car2 = new Car("Crysler", "4WD");
        Car.Engine c2engine = car2.new Engine();
        c2engine.setEngine();
        System.out.println("Engine Type for 4WD = " + c2engine.getEngineType());
    }
}
```

When you run above program, the output will be:

```
Engine Type for 8WD= Bigger
Engine Type for 4WD = Smaller
```

In above program, inside the Engine inner class, we used this keyword to get access to the member variable carType of outer class Car as:

```
Car.this.carType.equals("4WD)
```

This is possible even though the carType is a private member of Car class.
You can also see, we've used Car.this to access members of Car. If you had only used this instead of Car.this, then it would only represent members inside the Engine class.
Similarly, we've also invoked method getCarName() from Car using this keyword as:

```
Car.this.getCarName().equals("Crysler")
```

Here, getCarName() method is a private method of Car.

Static Inner Class:
In Java, you can also define a nested class static. Such class is known as static nested class. However, they are not called static inner class.
Unlike inner class, static nested class cannot access the member variables of the outer class because static nested class doesn't require you to create an instance of outer class. Hence, no reference of the outer class exists with OuterClass.this.

So, you can create instance of static nested class directly like this:
OuterClass.InnerClass obj = new OuterClass.InnerClass();

Example 3: Static Inner Class

```
Public class MotherBoard {
   String model;
   public MotherBoard(String model) {
      this.model = model;
   }
   static class USB{
      int usb2 = 2;
      int usb3 = 1;
      int getTotalPorts(){
         return usb2 + usb3;
      }
   }
}
public class Main {
   public static void main(String[] args) {
      MotherBoard.USB usb = new MotherBoard.USB();
      System.out.println("Total Ports = " + usb.getTotalPorts());
   }
}
```

When you run the above program, the output will be:

```
Total Ports = 3
```

In the above program, we've declared static inner class USB using the keyword static.
You can also see, in the Main class, we directly created an instance of USB from MotherBoard with the . (dot) operator without creating an instance of Motherboard first.
MotherBoard.USB usb = new MotherBoard.USB();
Let's see what would happen, if you try to access the members of the outer class:

Example 4: Accessing members of Outer class inside Static Inner Class

```
public class MotherBoard {
   String model;
   public MotherBoard(String model) {
      this.model = model;
   }
   static class USB{
      int usb2 = 2;
      int usb3 = 1;
      int getTotalPorts(){
         if(MotherBoard.this.model.equals("MSI")) {
            return 4;
         }
         else {
            return usb2 + usb3;
         }
      }
   }
}
```

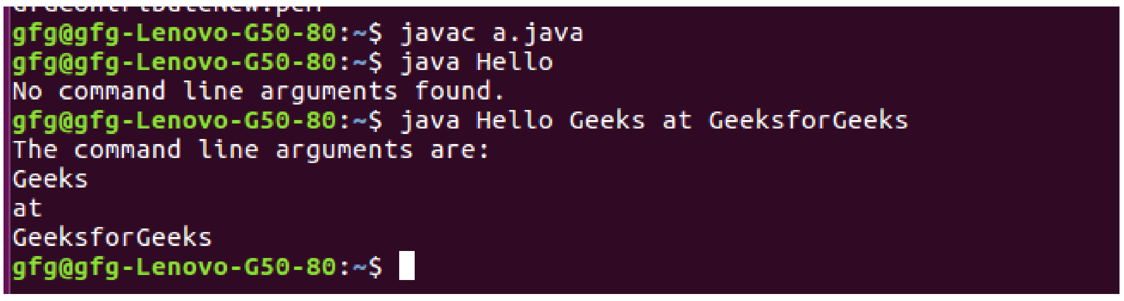
```
public class Main {
   public static void main(String[] args) {
      MotherBoard.USB usb = new MotherBoard.USB();
      System.out.println("Total Ports = " + usb.getTotalPorts());
   }
}
```

When you run above program, you'll get an error:
error: non-static variable this cannot be referenced from a static context

Using command line arguments: If we run a Java Program by writing the command "**java Hello Geeks At GeeksForGeeks**" where the name of the class is "Hello", then it will run upto Hello. It is command upto "Hello" and after that i.e "Geeks At GeeksForGeeks", these are command line arguments.When command line arguments are supplied to JVM, JVM wraps these and supply to args[]. It can be confirmed that they are actually wrapped up in args array by checking the length of args using args.length

```
// Program to check for command line arguments
class Hello
{
   public static void main(String[] args)
   {
      // check if length of args array is
      // greater than 0
      if (args.length > 0)
      {
         System.out.println("The command line"+
                        " arguments are:");

         // iterating the args array and printing
         // the command line arguments
         for (String val:args)
            System.out.println(val);
      }
      else
         System.out.println("No command line "+
                        "arguments found.");
   }
}
```

Output:



```
gfg@gfg-Lenovo-G50-80:~$ javac a.java
gfg@gfg-Lenovo-G50-80:~$ java Hello
No command line arguments found.
gfg@gfg-Lenovo-G50-80:~$ java Hello Geeks at GeeksforGeeks
The command line arguments are:
Geeks
at
GeeksforGeeks
gfg@gfg-Lenovo-G50-80:~$
```

2.6 INHERITANCE

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system). The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in java by which one class is allow to inherit the features(fields and methods) of another class.

2.7 INHERITANCE BASICS, USING SUPER, METHOD OVERRIDING, DYNAMIC METHOD DISPATCH

Important terminology:

- **Super Class:** The class whose features are inherited is known as super class(or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as sub class(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

How to use inheritance in Java

The keyword used for inheritance is extends.

Syntax :

```
class derived-class extends base-class
{
  //methods and fields
}
```

Example: In below example of inheritance, class Bicycle is a base class, class MountainBike is a derived class which extends Bicycle class and class Test is a driver class to run program.

```
//Java program to illustrate the
// concept of inheritance

// base class
class Bicycle
{
    // the Bicycle class has two fields
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int gear, int speed)
    {
        this.gear = gear;
        this.speed = speed;
    }

    // the Bicycle class has three methods
    public void applyBrake(int decrement)
    {
        speed -= decrement;
    }

    public void speedUp(int increment)
    {
        speed += increment;
    }

    // toString() method to print info of Bicycle
    public String toString()
    {
        return("No of gears are "+gear
                +"\n"
                + "speed of bicycle is "+speed);
    }
}

// derived class
class MountainBike extends Bicycle
{

    // the MountainBike subclass adds one more field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int gear,int speed,
                        int startHeight)
    {
        // invoking base-class(Bicycle) constructor
        super(gear, speed);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one more method
    public void setHeight(int newValue)
    {
        seatHeight = newValue;
    }

    // overriding toString() method
    // of Bicycle to print more info
    @Override
    public String toString()
    {
        return (super.toString()+
                "\nseat height is "+seatHeight);
    }

}

// driver class
public class Test
{
    public static void main(String args[])
    {

        MountainBike mb = new MountainBike(3, 100, 25);
        System.out.println(mb.toString());

    }
}
```

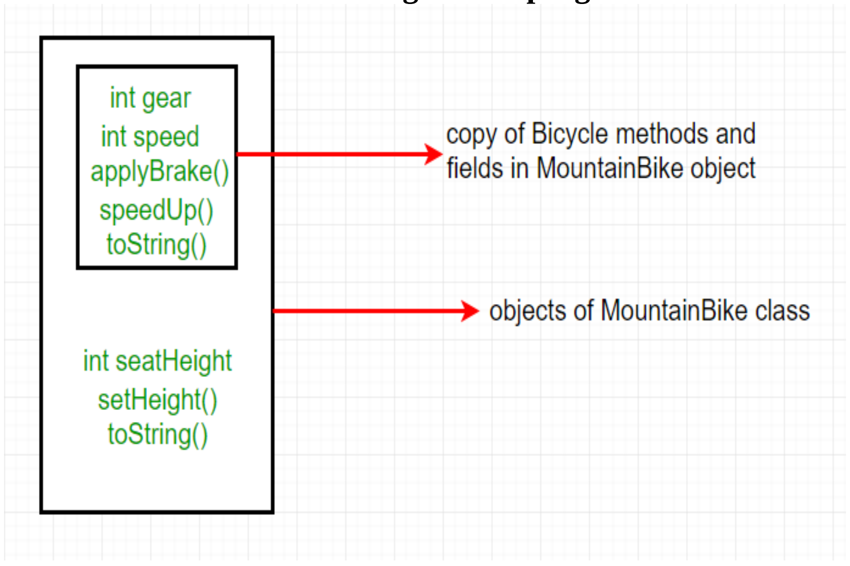
Output:

```
No of gears are 3
speed of bicycle is 100
seat height is 25
```

In above program, when an object of MountainBike class is created, a copy of the all methods and fields of the superclass acquire memory in this object. That is why; by using the object of the subclass we can also access the members of a superclass.

Please note that during inheritance only object of subclass is created, not the superclass.

Illustrative image of the program:


int gear
int speed
applyBrake()
speedUp()
toString()
copy of Bicycle methods and fields in MountainBike object
objects of MountainBike class
int seatHeight
setHeight()
toString()

Using super keyword

- The super keyword is similar to this keyword. Following are the scenarios where the super keyword is used.
- It is used to differentiate the members of superclass from the members of subclass, if they have same names.
- It is used to invoke the superclass constructor from subclass.

Differentiating the Members

If a class is inheriting the properties of another class. And if the members of the superclass have the names same as the sub class, to differentiate these variables we use super keyword as shown below.
super.variable
super.method();

Sample Code

This section provides you a program that demonstrates the usage of the super keyword.
In the given program, you have two classes namely *Sub_class* and *Super_class*, both have a method named display() with different implementations, and a variable named num with different values. We are invoking display() method of both classes and printing the value of the variable num of both classes. Here you can observe that we have used super keyword to differentiate the members of superclass from subclass.

Copy and paste the program in a file with name Sub_class.java.

Example

```
class Super_class {
   int num = 20;

   // display method of superclass
   public void display() {
      System.out.println("This is the display method of superclass");
   }
}
public class Sub_class extends Super_class {
   int num = 10;
   // display method of sub class
   public void display() {
      System.out.println("This is the display method of subclass");
   }
   public void my_method() {
      // Instantiating subclass
      Sub_class sub = new Sub_class();

      // Invoking the display() method of sub class
      sub.display();

      // Invoking the display() method of superclass
      super.display();

      // printing the value of variable num of subclass
      System.out.println("value of the variable named num in sub class:"+ sub.num);
      // printing the value of variable num of superclass
      System.out.println("value of the variable named num in super class:"+ super.num);
   }
   public static void main(String args[]) {
      Sub_class obj = new Sub_class();
      obj.my_method();
   }
}
```

Compile and execute the above code using the following syntax.

```
javac Super_Demo
java Super
```

On executing the program, you will get the following result −

Output

```
This is the display method of subclass
This is the display method of superclass
value of the variable named num in sub class:10
value of the variable named num in super class:20
```

method overriding-If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

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.

Usage of Java Method Overriding

Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.

Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

The method must have the same name as in the parent class

The method must have the same parameter as in the parent class.

There must be an IS-A relationship (inheritance).

```
//Java Program to demonstrate why we need method overriding
//Here, we are calling the method of parent class with child
```

```
//class object.
//Creating a parent class
class Vehicle{
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike extends Vehicle{
  public static void main(String args[]){
  //creating an instance of child class
  Bike obj = new Bike();
  //calling the method with child class instance
  obj.run();
  }
}
```

Output:

```
Vehicle is running
```

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
```

Dynamic method Dispatch- Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime. This is how java implements runtime polymorphism. When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to. In simple words the type of object which it referred determines which version of overridden method will be called.

upcasting in java

When Parent class reference variable refers to Child class object, it is known as Upcasting. In Java this can be done and is helpful in scenarios where multiple child classes extends one parent class. In those cases we can create a parent class reference and assign child class objects to it.

Let's take an example to understand it,

```
class Game
{
        public void type()
        {
                System.out.println("Indoor & outdoor");
        }
}

Class Cricket extends Game
{
        public void type()
        {
                System.out.println("outdoor game");
        }

        public static void main(String[] args)
        {
                Game gm = new Game();
                Cricket ck = new Cricket();
                gm.type();
                ck.type();
                gm = ck;        //gm refers to Cricket object
                gm.type();      //calls Cricket's version of type
        }
}
```

```
Indoor & outdoor
Outdoor game
Outdoor game
```

Notice the last output. This is because of the statement, gm = ck;. Now gm.type() will call the Cricket class

2.8 USING ABSTRACT CLASSES, USING FINAL WITH INHERITANCE

In C++, if a class has at least one pure virtual function, then the class becomes abstract. Unlike C++, in Java, a separate keyword *abstract* is used to make a class abstract.

```
// An example abstract class in Java
abstract class Shape {
   int color;

   // An abstract function (like a pure virtual function in C++)
   abstract void draw();
}
```

Following are some important observations about abstract classes in Java.

) Like C++, in Java, an instance of an abstract class cannot be created, we can have references of abstract class type though.

```
abstract class Base {
   abstract void fun();
}
class Derived extends Base {
   void fun() { System.out.println("Derived fun() called"); }
}
class Main {
   public static void main(String args[]) {

      // Uncommenting the following line will cause compiler error as the
      // line tries to create an instance of abstract class.
      // Base b = new Base();

      // We can have references of Base type.
      Base b = new Derived();
      b.fun();
   }
}
```

Output:

Derived fun() called

Like C++, an abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of a inherited class is created. For example, the following is a valid Java program.

```
       // An abstract class with constructor
       abstract class Base {
          Base() { System.out.println("Base Constructor Called"); }
          abstract void fun();
       }
       class Derived extends Base {
          Derived() { System.out.println("Derived Constructor Called"); }
          void fun() { System.out.println("Derived fun() called"); }
       }
       class Main {
          public static void main(String args[]) {
            Derived d = new Derived();
          }
       }
```

Output:

```
Base Constructor Called
Derived Constructor Called
```

In Java, we can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated, but can only be inherited.

```
// An abstract class without any abstract method
abstract class Base {
   void fun() { System.out.println("Base fun() called"); }
}

class Derived extends Base { }

class Main {
   public static void main(String args[]) {
      Derived d = new Derived();
      d.fun();
   }
}
```

Output:

```
Base fun() called
```

Abstract classes can also have final methods (methods that cannot be overridden). For example, the following program compiles and runs fine.

```
// An abstract class with a final method
abstract class Base {
   final void fun() { System.out.println("Derived fun() called"); }
}

class Derived extends Base {}

class Main {
   public static void main(String args[]) {
      Base b = new Derived();
      b.fun();
   }
}
```

Output:

```
Derived fun() called
```

using final with Inheritance- During inheritance, we must declare methods with final keyword for which we required to follow the same implementation throughout all the derived classes. Note that it is not necessary to declare final methods in the initial stage of inheritance(base class always). We can declare final method in any subclass for which we want that if any other class extends this subclass, then it must follow same implementation of the method as in the that subclass.

```
// Java program to illustrate
// use of final with inheritance

// base class
abstract class Shape
{
   private double width;

   private double height;

   // Shape class parameterized constructor
   public Shape(double width, double height)
   {
      this.width = width;
      this.height = height;
   }

   // getWidth method is declared as final
   // so any class extending
   // Shape cann't override it
   public final double getWidth()
   {
      return width;
   }
```

```
    // getHeight method is declared as final
    // so any class extending Shape
    // can not override it
    public final double getHeight()
    {
        return height;
    }


    // method getArea() declared abstract because
    // it upon its subclasses to provide
    // complete implementation
    abstract double getArea();
}

// derived class one
class Rectangle extends Shape
{
    // Rectangle class parameterized constructor
    public Rectangle(double width, double height)
    {
        // calling Shape class constructor
        super(width, height);
    }

    // getArea method is overridden and declared
    // as final    so any class extending
    // Rectangle cann't override it
    @Override
    final double getArea()
    {
        return this.getHeight() * this.getWidth();
    }

}
  //derived class two
class Square extends Shape
{
    // Square class parameterized constructor
    public Square(double side)
    {
        // calling Shape class constructor
        super(side, side);
    }

    // getArea method is overridden and declared as
    // final so any class extending
    // Square cann't override it
    @Override
    final double getArea()
    {
        return this.getHeight() * this.getWidth();
    }

}

// Driver class
public class Test
{
    public static void main(String[] args)
    {
        // creating Rectangle object
        Shape s1 = new Rectangle(10, 20);

        // creating Square object
        Shape s2 = new Square(10);

        // getting width and height of s1
        System.out.println("width of s1 : "+ s1.getWidth());
        System.out.println("height of s1 : "+ s1.getHeight());

        // getting width and height of s2
        System.out.println("width of s2 : "+ s2.getWidth());
        System.out.println("height of s2 : "+ s2.getHeight());

        //getting area of s1
        System.out.println("area of s1 : "+ s1.getArea());

        //getting area of s2
        System.out.println("area of s2 : "+ s2.getArea());

    }
}
```

Output:

```
width of s1 : 10.0
height of s1 : 20.0
width of s2 : 10.0
height of s2 : 10.0
area of s1 : 200.0
area of s2 : 100.0
```

Using final to Prevent Inheritance

When a class is declared as final then it cannot be subclassed i.e. no any other class can extend it. This is particularly useful, for example, when creating an immutable class like the predefined String class. The following fragment illustrates final keyword with a class:

```
final class A
{
     // methods and fields
}
// The following class is illegal.
class B extends A
```

```
{
   // ERROR! Can't subclass A
}
```

Note :

Declaring a class as final implicitly declares all of its methods as final, too.
It is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Using final to Prevent Overriding

When a method is declared as final then it cannot be overridden by subclasses.The Object class does this—a number of its methods are final. The following fragment illustrates final keyword with a method:

```
class A
{
   final void m1()
   {
      System.out.println("This is a final method.");
   }
}

class B extends A
{
   void m1()
   {
      // ERROR! Can't override.
      System.out.println("Illegal!");
   }
}
```

2.9 SUMMARY

Classes and Objects in Java. Classes and Objects are basic concepts of Object Oriented Programming which revolve around the real life entities. A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviours of a parent object. It is an important part of OOPs (Object Oriented programming system).The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also. Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

2.10 SELF ASSESMENT QUESTIONS

1. How objects is declared
2. Explain Argument passing
3. Explain Nested and Inner classes
4. Explain “this” keyword
5. Explain method overriding
6. Explain using final with Inheritance

7. Explain dynamic method Dispatch

2.11 SUGGESTED READING

1 The complete reference Java –2: V Edition By Herbert Schildt Pub. TMH.
2. SAMS teach yourself Java – 2: 3rd Edition by Rogers Cedenhead and Leura Lemay Pub. Pearson Education
3. Java: A Beginner's Guide by Herbert Schildt
4. Java: Programming Basics for Absolute Beginners by Nathan Clark.