Important Topics (from chapter 3 & 4)

1. Object & Class

Object − Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.

Objects in Java

Let us now look deep into what are objects. If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.

If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running.

If you compare the software object with a real-world object, they have very similar characteristics.

Software objects also have a state and a behavior. A software object's state is stored in fields and behavior is shown via methods.

So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

Classes in Java

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:

1. **Modifiers** : A class can be public or has default access (Refer this for details).
2. **Class name:** The name should begin with a initial letter (capitalized by convention).
3. **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.
4. **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.
5. **Body:** The class body surrounded by braces, { }.

Constructors are used for initializing new objects.

Fields are variables that provides the state of the class and its objects,

and methods are used to implement the behavior of the class and its objects.

Following is a sample of a class.

Example

```
public class Dog {
   String breed;
   int age;
   String color;

   void barking() {
   }

   void hungry() {
   }

   void sleeping() {
   }
}
```

A class can contain any of the following variable types.

- **Local variables** − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables** − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class variables** − Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

Following are some of the important topics that need to be discussed when looking into classes of the Java Language.

Constructors

When discussing about classes, one of the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Following is an example of a constructor −

Example

```
public class Puppy {
   public Puppy() {
   }

   public Puppy(String name) {
      // This constructor has one parameter, name.
   }
}
```

Java also supports Singleton Classes where you would be able to create only one instance of a class.

Note − We have two different types of constructors. We are going to discuss constructors in detail in the subsequent chapters.

Creating an Object

As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.

There are three steps when creating an object from a class −

- **Declaration** − A variable declaration with a variable name with an object type.
- **Instantiation** − The 'new' keyword is used to create the object.
- **Initialization** − The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Following is an example of creating an object −

Example

```
public class Puppy {
   public Puppy(String name) {
      // This constructor has one parameter, name.
      System.out.println("Passed Name is :" + name );
   }

   public static void main(String []args) {
      // Following statement would create an object myPuppy
      Puppy myPuppy = new Puppy( "tommy" );
   }
}
```

If we compile and run the above program, then it will produce the following result −

Output

```
Passed Name is :tommy
```

2. Difference between private, protected, public keyword or modifiers in JAVA

private vs public vs protected vs package in Java

Java has four access modifier namely `private`, `protected` and `public`. package level access is **default access** level provided by Java if no access modifier is specified. These access modifiers are used to restrict accessibility of a class, method or variable on which it applies. We will start from private access modifier which is most restrictive access modifier and then go towards public which is least restrictive access modifier, along the way we will see some best practices while using access modifier in Java and some examples of using `private` and `protected` keywords.

private keyword in Java

`private` keyword or modifier in java can be applied to member field, method or nested class in Java. you can not use the `private` modifier on top level class. `private` variables, methods, and class are only accessible on the class on which they are declared. `private` is the highest form of Encapsulation Java API provides and should be used as much as possible. It's best coding practice in Java to declare variable private by default. `a private` method can only be called from the class where it has declared.

As per Rules of method overriding in Java, a `private` method can not be overridden as well. `the private` keyword can also be applied to the constructor and if you make constructor private you prevent it from being sub-classed. a popular example of making the constructor private is Singleton class in Java which provides getInstance() method to get object instead of creating a new object using the constructor in Java. here are some differences between `private` and `protected`, `public` and package level access.

package or default access level in Java

there is no access modifier called package instead `package` is a keyword which is used to declare a package in Java, a package is a directory on which a class in Java belongs. Package or default access level is second highest restrictive access modifier after `private` and any variable, method or class declared as `package-private` is only accessible on the package it belongs. the good thing about default modifier is that top level class can also be package-private if there is no class level access modifier.

protected keyword in Java

The difference between `private` and `protected` keyword is that protected method, variable or nested class not only accessible inside a class, inside the package but also outside of package on a subclass. if you declare a variable `protected` means anyone can use it if they extend your class. the top level class can not be make `protected` as well.

public keyword in Java

`public` is the least restrictive access modifier in Java programming language and its bad practice to declare field, method or class by default public because once you make it public it's very difficult to make any change on the internal structure of class as it affects all clients using it. Making class or instance variable public also violated the principle of Encapsulation which is not good at all and affects maintenance badly. instead of making variable `public` you should make it `private` and provided public getter and setter. `the public` modifier can also be applied to a top-level class. In Java name of the file must be same with public class declared in the file.

That's all difference between `private`, `protected`, `package` and `public` access modifier. As you have seen the difference between private and public lies on how accessible a particular field, method or class would have. public means you can access it anywhere while private means you can only access it inside its own class.

3. Types of References in Java

In Java there are four types of references differentiated on the way by which they are garbage collected.

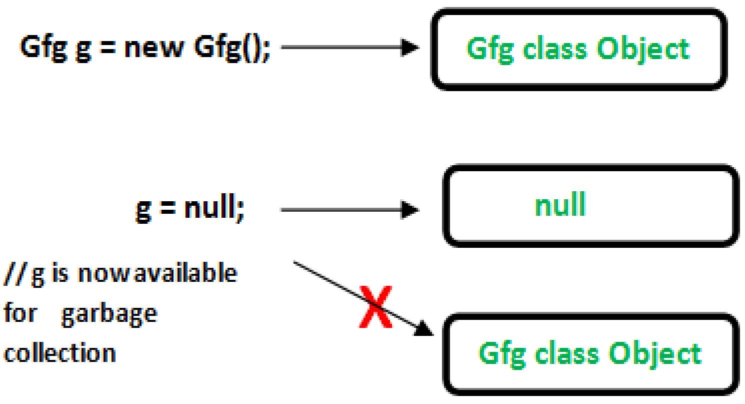
1. Strong References
2. Weak References
3. Soft References
4. Phantom References

Prerequisite: Garbage Collection

- **Strong References:** This is the default type/class of Reference Object. Any object which has an active strong reference are not eligible for garbage collection. The object is garbage collected only when the variable which was strongly referenced points to null.
- `MyClass obj = new MyClass ();`

  Here 'obj' object is strong reference to newly created instance of MyClass, currently obj is active object so can't be garbage collected.

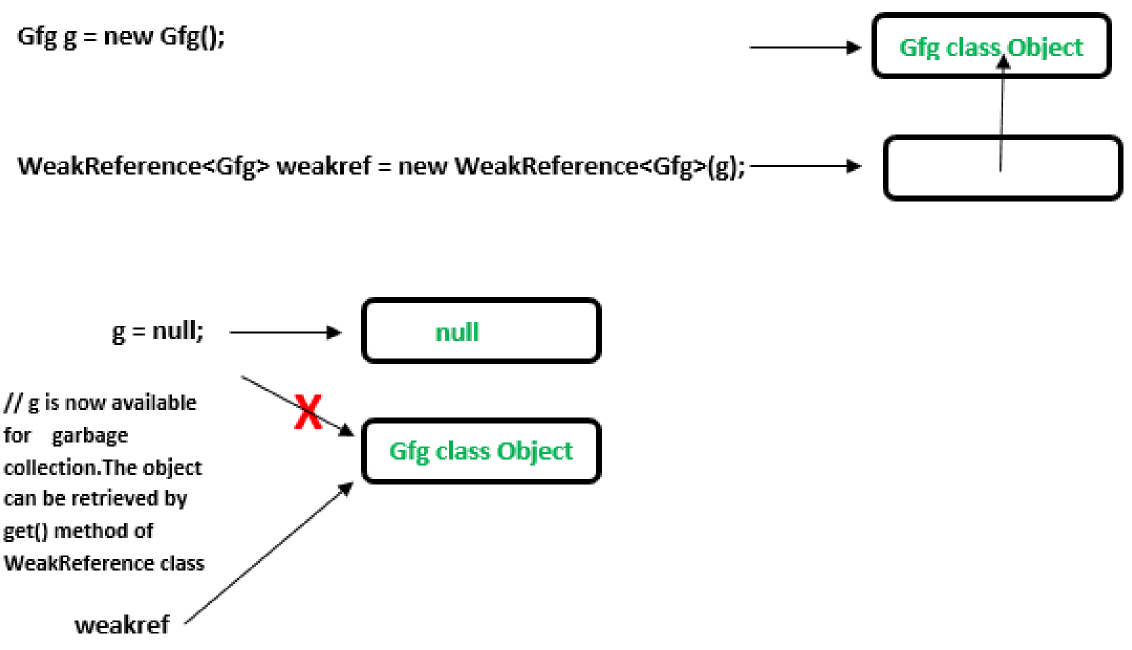
```
obj = null;
//'obj' object is no longer referencing to the instance.
So the 'MyClass type object is now available for garbage
collection.
```



```
// Java program to illustrate Strong reference
class Gps
{
    //Code..
}
```

```
public class Example
{
    public static void main(String[] args)
    {
         //Strong Reference - by default
        Gps g = new Gps();

        //Now, object to which 'g' was pointing earlier is
        //eligible for garbage collection.
        g = null;
    }
}
```

- **Weak References:** Weak Reference Objects are not the default type/class of Reference Object and they should be explicitly specified while using them.
    - This type of reference is used in WeakHashMap to reference the entry objects .
    - If JVM detects an object with only weak references (i.e. no strong or soft references linked to any object object), this object will be marked for garbage collection.
    - To create such references java.lang.ref.WeakReference class is used.
    - These references are used in real time applications while establishing a DBConnection which might be cleaned up by Garbage Collector when the application using the database gets closed.


Gfg g = new Gfg();
Gfg class Object
WeakReference<Gfg> weakref = new WeakReference<Gfg>(g);
g = null;
null
X
// g is now available for garbage collection.The object can be retrieved by get() method of WeakReference class
Gfg class Object
weakref

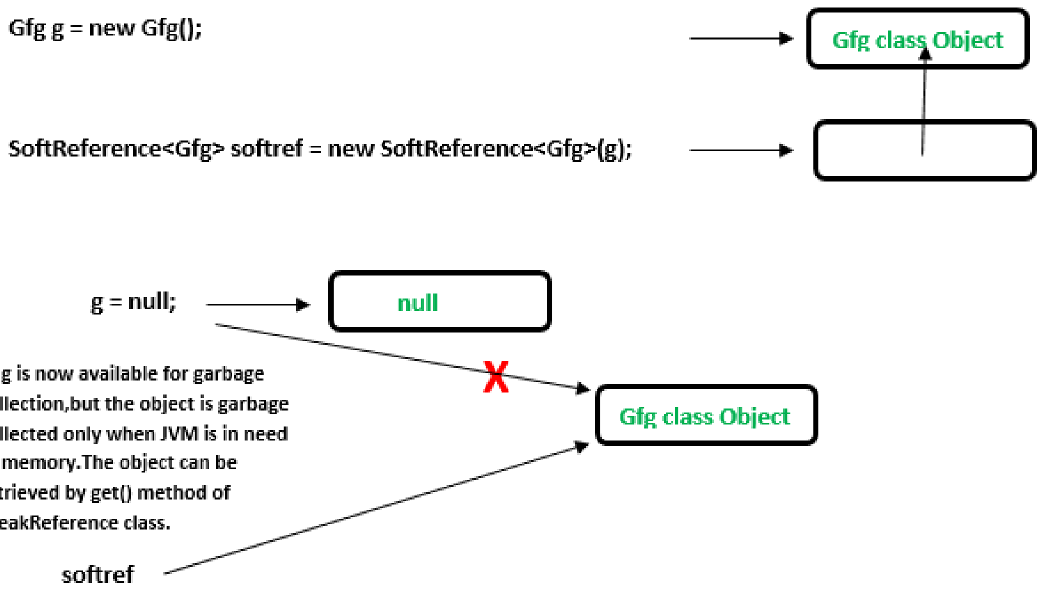
```
//Java Code to illustrate Weak reference
import java.lang.ref.WeakReference;
class Gps
{
    //code
    public void x()
    {
        System.out.println("GovtPolytehnicSonipat");
    }
}
 public class Example
{
    public static void main(String[] args)
    {
        // Strong Reference
        Gps g = new Gps();
        g.x();
           // Creating Weak Reference to Gps-type object to which 'g'
        // is also pointing.
        WeakReference<Gps> weakref = new WeakReference<Gps>(g);

        //Now, Gps-type object to which 'g' was pointing earlier
        //is available for garbage collection.
        //But, it will be garbage collected only when JVM needs memory.
        g = null;

        // You can retrieve back the object which
        // has been weakly referenced.
        // It successfully calls the method.
        g = weakref.get();

        g.x();
    }
}
```

Output:

```
GovtPolitechnicSonipat
GovtPolytechnicSonipat
```

Two different levels of weakness can be enlisted: Soft and Phantom

- **Soft References:** In Soft reference, even if the object is free for garbage collection then also its not garbage collected, until JVM is in need of memory badly.The objects gets cleared from the memory when JVM runs out of memory.To create such references java.lang.ref.SoftReference class is used.


Gfg g = new Gfg();
Gfg class Object
SoftReference<Gfg> softref = new SoftReference<Gfg>(g);
g = null;
null
// g is now available for garbage collection,but the object is garbage collected only when JVM is in need of memory.The object can be retrieved by get() method of WeakReference class.
X
Gfg class Object
softref

```
//Code to illustrate Soft reference
import java.lang.ref.SoftReference;
class Gps
{
    //code..
    public void x()
    {
        System.out.println("GovtPolytechnicSonipat");
    }
}

public class Example
{
    public static void main(String[] args)
    {
        // Strong Reference
        Gps g = new Gps();
        g.x();
        // Creating Soft Reference to Gps-type object to which 'g'
        // is also pointing.
        SoftReference<Gps> softref = new SoftReference<Gps>(g);

        // Now, Gps-type object to which 'g' was pointing
        // earlier is available for garbage collection.
        g = null;

        // You can retrieve back the object which
        // has been weakly referenced.
        // It successfully calls the method.
        g = softref.get();

        g.x();
    }
}
```

Output:

```
GovtPolytechnicSonipat
GovtPolytechnicSonipat
```

Chapter-4

Inheritance (Modifiers)

Java Access Modifiers – Public, Private, Protected & Default

You must have seen public, private and protected keywords while practising java programs, these are called access modifiers. An access modifier restricts the access of a class, constructor, data member and method in another class. In java we have four access modifiers:
1. default
2. private
3. protected
4. public

1. Default access modifier

When we do not mention any access modifier, it is called default access modifier. The scope of this modifier is limited to the package only. This means that if we have a class with the default access modifier in a package, only those classes that are in this package can access this class. No other class outside this package can access this class. Similarly, if we have a default method or data member in a class, it would not be visible in the class of another package. Lets see an example to understand this:

Default Access Modifier Example in Java

To understand this example, you must have the knowledge of **packages in java**.

In this example we have two classes, Test class is trying to access the default method of Addition class, since class Test belongs to a different package, this program would throw compilation error, because the scope of default modifier is limited to the same package in which it is declared.
Addition.java

```
package abcpackage;

public class Addition {
   /* Since we didn't mention any access modifier here, it would
    * be considered as default.
    */
   int addTwoNumbers(int a, int b){
        return a+b;
   }
}
```

Test.java

```
package xyzpackage;

/* We are importing the abcpackage
 * but still we will get error because the
 * class we are trying to use has default access
 * modifier.
 */
import abcpackage.*;
public class Test {
   public static void main(String args[]){
        Addition obj = new Addition();
         /* It will throw error because we are trying to access
          * the default method in another package
          */
        obj.addTwoNumbers(10, 21);
   }
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The method addTwoNumbers(int, int) from the type Addition is not visible
at xyzpackage.Test.main(Test.java:12)
```

2. Private access modifier

The scope of private modifier is limited to the class only.

1. Private Data members and methods are only accessible within the class
2. Class and **Interface** cannot be declared as private
3. If a class has **private constructor** then you cannot create the object of that class from outside of the class.

Let’s see an example to understand this:

Private access modifier example in java

This example throws compilation error because we are trying to access the private data member and method of class ABC in the class Example. The private data member and method are only accessible within the class.

```
class ABC{
   private double num = 100;
   private int square(int a){
        return a*a;
   }
}
public class Example{
   public static void main(String args[]){
        ABC obj = new ABC();
        System.out.println(obj.num);
        System.out.println(obj.square(10));
   }
}
```

Output:

```
Compile - time error
```

3. Protected Access Modifier

Protected data member and method are only accessible by the classes of the same package and the subclasses present in any package. You can also say that the protected access modifier is similar to default access modifier with one exception that it has visibility in sub classes.
Classes cannot be declared protected. This access modifier is generally used in a parent child relationship.

Protected access modifier example in Java

In this example the class Test which is present in another package is able to call the `addTwoNumbers()` method, which is declared protected. This is because the Test class extends class Addition and the protected modifier allows the access of protected members in subclasses (in any packages).

Addition.java

```
package abcpackage;
public class Addition {

   protected int addTwoNumbers(int a, int b){
        return a+b;
   }
}
```

Test.java

```
package xyzpackage;
import abcpackage.*;
class Test extends Addition{
   public static void main(String args[]){
        Test obj = new Test();
        System.out.println(obj.addTwoNumbers(11, 22));
   }
}
```

Output:

```
33
```

4. Public access modifier

The members, methods and classes that are declared public can be accessed from anywhere. This modifier doesn't put any restriction on the access.

public access modifier example in java

Lets take the same example that we have seen above but this time the method addTwoNumbers() has public modifier and class Test is able to access this method without even extending the Addition class. This is because public modifier has visibility everywhere.

Addition.java

```
package abcpackage;

public class Addition {

   public int addTwoNumbers(int a, int b){
        return a+b;
   }
}
```

Test.java

```
package xyzpackage;
import abcpackage.*;
class Test{
   public static void main(String args[]){
      Addition obj = new Addition();
      System.out.println(obj.addTwoNumbers(100, 1));
   }
}
```

Output:

```
101
```

Lets see the scope of these access modifiers in tabular form:

The scope of access modifiers in tabular form

	Class	Package	Subclass (same package)	Subclass (diff package)	Outside Class
public	Yes	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	Yes	No
default	Yes	Yes	Yes	No	No
private	Yes	No	No	No	No

Constructors in Java

Constructors are used to initialize the object's state. Like **methods**, a constructor also contains **collection of statements(i.e. instructions)** that are executed at time of Object creation.

Need of Constructor
Think of a Box. If we talk about a box class then it will have some class variables (say length, breadth, and height). But when it comes to creating its object(i.e Box will now exist in computer's memory), then can a box be there with no value defined for its dimensions. The answer is no.
So constructors are used to assign values to the class variables at the time of object creation, either explicitly done by the programmer or by Java itself (default constructor).

When is a Constructor called ?
Each time an object is created using **new()** keyword at least one constructor (it could be default constructor) is invoked to assign initial values to the **data members** of the same class.

A constructor is invoked at the time of object or instance creation. For Example:

```
class Geek

{

  .......

  // A Constructor

  new Geek() {}

  .......

}
```

```
// We can create an object of the above class

// using the below statement. This statement

// calls above constructor.

Geek obj = new Geek();
```

Rules for writing Constructor:

- Constructor(s) of a class must has same name as the class name in which it resides.
- A constructor in Java can not be abstract, final, static and Synchronized.
- Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

Types of constructor

There are two type of constructor in Java:

1. **No-argument constructor:** A constructor that has no parameter is known as default constructor. If we don't define a constructor in a class, then compiler creates **default constructor(with no arguments)** for the class. And if we write a constructor with arguments or no-arguments then the compiler does not create a default constructor.
   Default constructor provides the default values to the object like 0, null, etc. depending on the type.

   filter_none
   edit

   play_arrow

   brightness_4

```
// Java Program to illustrate calling a


// no-argument constructor


import java.io.*;


  class Geek


{
    int num;

    String name;

      // this would be invoked while an object

    // of that class is created.

    Geek()

    {

        System.out.println("Constructor called");

    }

}

  class GFG

{

    public static void main (String[] args)

    {

        // this would invoke default constructor.

        Geek geek1 = new Geek();
```

```
          // Default constructor provides the default

        // values to the object like 0, null

        System.out.println(geek1.name);

        System.out.println(geek1.num);

    }

}
```

Output :

```
Constructor called

null

0
```

2. **Parameterized Constructor:** A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with your own values, then use a parameterized constructor.

filter_none

edit

play_arrow

brightness_4

```
// Java Program to illustrate calling of

// parameterized constructor.

import java.io.*;
```

```
  class Geek

{

    // data members of the class.

    String name;

    int id;

      // constructor would initialize data members

    // with the values of passed arguments while

    // object of that class created.

    Geek(String name, int id)

    {

        this.name = name;

        this.id = id;

    }

}

  class GFG
```

```
{

    public static void main (String[] args)

    {

        // this would invoke the parameterized constructor.

        Geek geek1 = new Geek("adam", 1);

        System.out.println("GeekName :" + geek1.name +

                                   " and GeekId :" + geek1.id);

    }

}
```

Output:

```
GeekName :adam and GeekId :1
```

Does constructor return any value?

There are no “return value” statements in constructor, but constructor returns current class instance. We can write ‘return’ inside a constructor.

Constructor Overloading

Like methods, we can overload constructors for creating objects in different ways. Compiler differentiates constructors on the basis of numbers of parameters, types of the parameters and order of the parameters.

filter_none
edit

play_arrow

brightness_4

```
// Java Program to illustrate constructor overloading

// using same task (addition operation ) for different

// types of arguments.

import java.io.*;

  class Geek

{

    // constructor with one argument

    Geek(String name)

    {

        System.out.println("Constructor with one " +

                      "argument - String : " + name);

    }

      // constructor with two arguments

    Geek(String name, int age)

    {
        System.out.println("Constructor with two arguments : " +

                " String and Integer : " + name + " "+ age);

      }

      // Constructor with one argument but with different

    // type than previous..

    Geek(long id)

    {

        System.out.println("Constructor with one argument : " +

                                            "Long : " + id);

    }

}

  class GFG

{

    public static void main(String[] args)

    {
        // Creating the objects of the class named 'Geek'

        // by passing different arguments

          // Invoke the constructor with one argument of

        // type 'String'.

        Geek geek2 = new Geek("Shikhar");

          // Invoke the constructor with two arguments

        Geek geek3 = new Geek("Dharmesh", 26);

          // Invoke the constructor with one argument of

        // type 'Long'.

        Geek geek4 = new Geek(325614567);

    }

}
```

Output:

```
Constructor with one argument - String : Shikhar

Constructor with two arguments - String and Integer : Dharmesh 26

Constructor with one argument - Long : 325614567
```

Constructor chaining

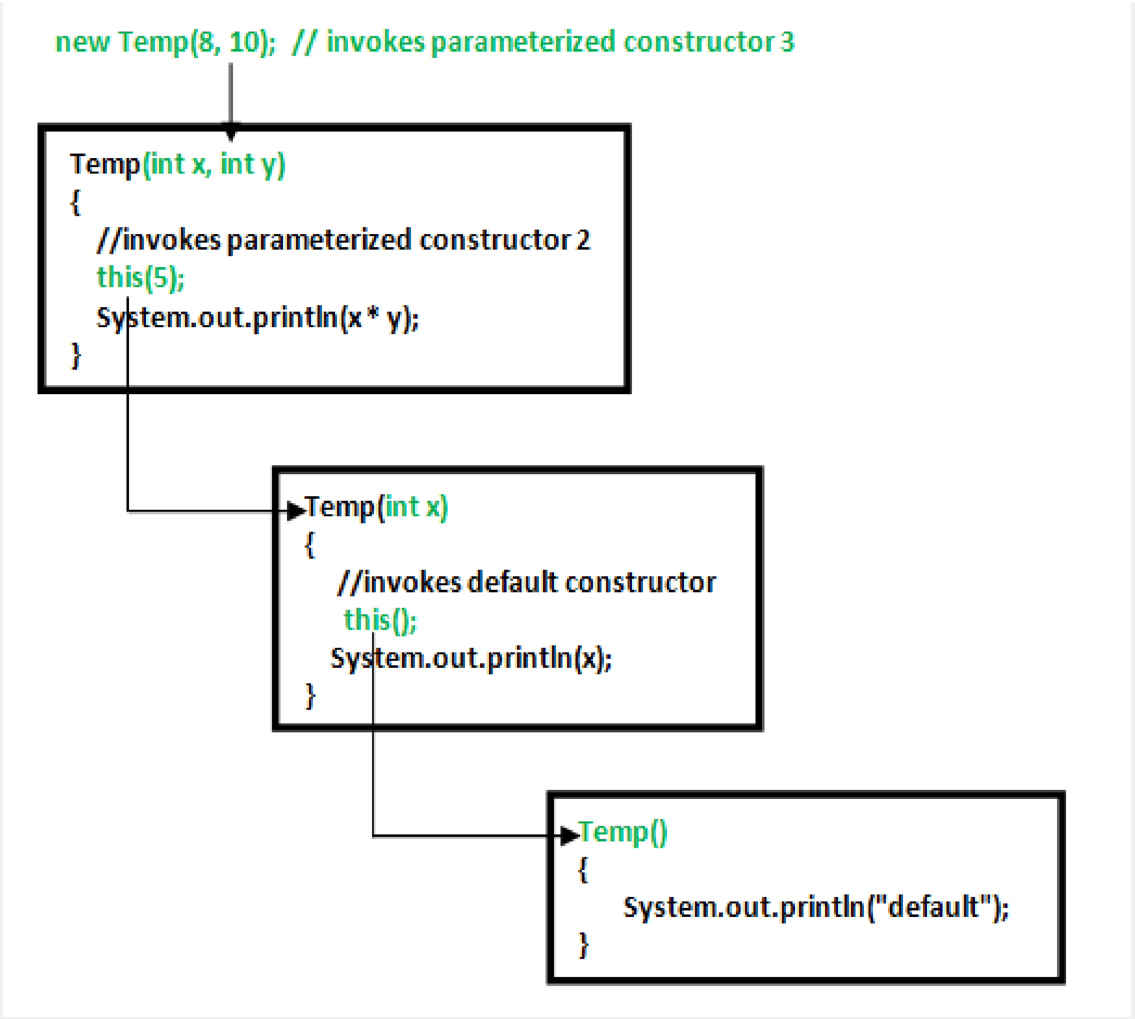
Constructor chaining is the process of calling one constructor from another constructor with respect to current object.
Constructor chaining can be done in two ways:

- **Within same class**: It can be done using **this()** keyword for constructors in same class
- **From base class:** by using **super()** keyword to call constructor from the base class.

Constructor chaining occurs through **inheritance**. A sub class constructor's task is to call super class's constructor first. This ensures that creation of sub class's object starts with the initialization of the data members of the super class. There could be any numbers of classes in inheritance chain. Every constructor calls up the chain till class at the top is reached.

Why do we need constructor chaining ?
This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable.

Constructor Chaining within same class using this() keyword :


new Temp(8, 10); // invokes parameterized constructor 3
Temp(int x, int y)
{
//invokes parameterized constructor 2
this(5);
System.out.println(x * y);
}
Temp(int x)
{
//invokes default constructor
this();
System.out.println(x);
}
Temp()
{
System.out.println("default");
}

filter_none
edit

play_arrow

brightness_4

```
// Java program to illustrate Constructor Chaining


// within same class Using this() keyword


class Temp
```

```
{

    // default constructor 1

    // default constructor will call another constructor

    // using this keyword from same class

    Temp()

    {

        // calls constructor 2

        this(5);

        System.out.println("The Default constructor");

    }

      // parameterized constructor 2

    Temp(int x)

    {

        // calls constructor 3

        this(5, 15);

        System.out.println(x);
    }

      // parameterized constructor 3

    Temp(int x, int y)

    {

        System.out.println(x * y);

    }

      public static void main(String args[])

    {

        // invokes default constructor first

        new Temp();

    }

}
```

Output:

```
75

5

The Default constructor
```

Rules of constructor chaining :

1. The **this()** expression should always be the first line of the constructor.
2. There should be at-least be one constructor without the this() keyword (constructor 3 in above example).

3. Constructor chaining can be achieved in any order.

What happens if we change the order of constructors?

Nothing, Constructor chaining can be achieved in any order

filter_none
edit

play_arrow

brightness_4

```
// Java program to illustrate Constructor Chaining

// within same class Using this() keyword

// and changing order of constructors

class Temp

{

    // default constructor 1

    Temp()

    {

        System.out.println("default");

    }

      // parameterized constructor 2
```

```
Temp(int x)

{

    // invokes default constructor

    this();

    System.out.println(x);

}

  // parameterized constructor 3

Temp(int x, int y)

{

    // invokes parameterized constructor 2

    this(5);

    System.out.println(x * y);

}




public static void main(String args[])

{
        // invokes parameterized constructor 3

        new Temp(8, 10);

    }

}
```

Output:

```
default

5

80
```

NOTE: In example 1, default constructor is invoked at the end, but in example 2 default constructor is invoked at first. Hence, order in constructor chaining is not important.

Constructor Chaining to other class using super() keyword :

filter_none
edit

play_arrow

brightness_4

```
// Java program to illustrate Constructor Chaining to

// other class using super() keyword

class Base

{

    String name;
```

```
     // constructor 1

    Base()

    {

        this("");

        System.out.println("No-argument constructor of" +

                                       " base class");

    }

     // constructor 2

    Base(String name)

    {

        this.name = name;

        System.out.println("Calling parameterized constructor"

                                          + " of base");

    }

}
```

```
  class Derived extends Base

{

    // constructor 3

    Derived()

    {

        System.out.println("No-argument constructor " +

                           "of derived");

    }

      // parameterized constructor 4

    Derived(String name)

    {

        // invokes base class constructor 2

        super(name);

        System.out.println("Calling parameterized " +

                           "constructor of derived");

    }
```

```
      public static void main(String args[])

    {

        // calls parameterized constructor 4

        Derived obj = new Derived("test");

          // Calls No-argument constructor

        // Derived obj = new Derived();

    }

}
```

Output:

```
Calling parameterized constructor of base

Calling parameterized constructor of derived
```

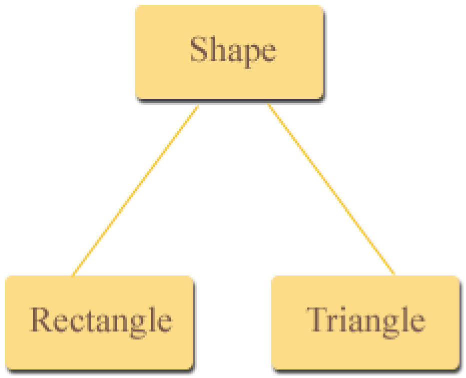
9.4 Protected Member

The private members of a class cannot be directly accessed outside the class. Only methods of that class can access the private members directly. As discussed previously, however, sometimes it may be necessary for a subclass to access a private member of a superclass. If you make a private member public, then anyone can access that member. So, if a member of a superclass needs to be (directly) accessed in a subclass and yet still prevent its direct access outside the class, you must declare that member **protected**.
Following table describes the difference

Modifier	Class	Subclass	World

public	Y	Y	Y
protected	Y	Y	N
private	Y	N	N

Following program illustrates how the methods of a subclass can directly access a protected member of the superclass.



For example, let's imagine a series of classes to describe two kinds of shapes: rectangles and triangles. These two shapes have certain common properties height and a width (or base).

This could be represented in the world of classes with a class Shapes from which we would derive the two other ones : Rectangle and Triangle

Program : (Shape.java)

```
/**
 *  A class Shape that holds width and height
 *  of any shape
 */
public class Shape
{
   protected double height;  // To hold height.
   protected double width;  //To hold width or base

   /**
    *  The setValue method sets the data
    *  in the height and width field.
    */
```

```java
   public void setValues(double height, double width)
   {
      this.height = height;
      this.width = width;
   }
}
```

Program : (Rectangle.java)

```java
/**
 *  This class Rectangle calculates
 *  the area of rectangle
 */
public class Rectangle extends Shape
{

   /**
    * The method returns the area
    * of rectangle.
    */
   public double getArea()
   {
      return height * width; //accessing protected members
   }
}
```

Program : (Triangle.java)

```java
/**
 *  This class Triangle calculates
 *  the area of triangle
 */
public class Triangle extends Shape
{

   /**
    * The method returns the area
    * of triangle.
    */
   public double getArea()
   {
      return height * width / 2; //accessing protected members
   }
}
```

Program : (TestProgram.java)

```
/**
 *  This program demonstrates the Rectangle and
 *  Triangle class, which inherits from the Shape class.
 */
public class TestProgram
{
   public static void main(String[] args)
   {
      //Create object of Rectangle.
      Rectangle rectangle = new Rectangle();

      //Create object of Triangle.
      Triangle triangle = new Triangle();

      //Set values in rectangle object
      rectangle.setValues(5,4);

      //Set values in trianlge object
      triangle.setValues(5,10);

      // Display the area of rectangle.
      System.out.println("Area of rectangle : " +
                         rectangle.getArea());

      // Display the area of triangle.
      System.out.println("Area of triangle : " +
                         triangle.getArea());
   }
}
```

Output :

```
Area of rectangle : 20.0
Area of triangle : 25.0
```

Inheritance in Java

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.
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.

filter_none
edit
play_arrow
brightness_4

```
//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);

    }

}
```

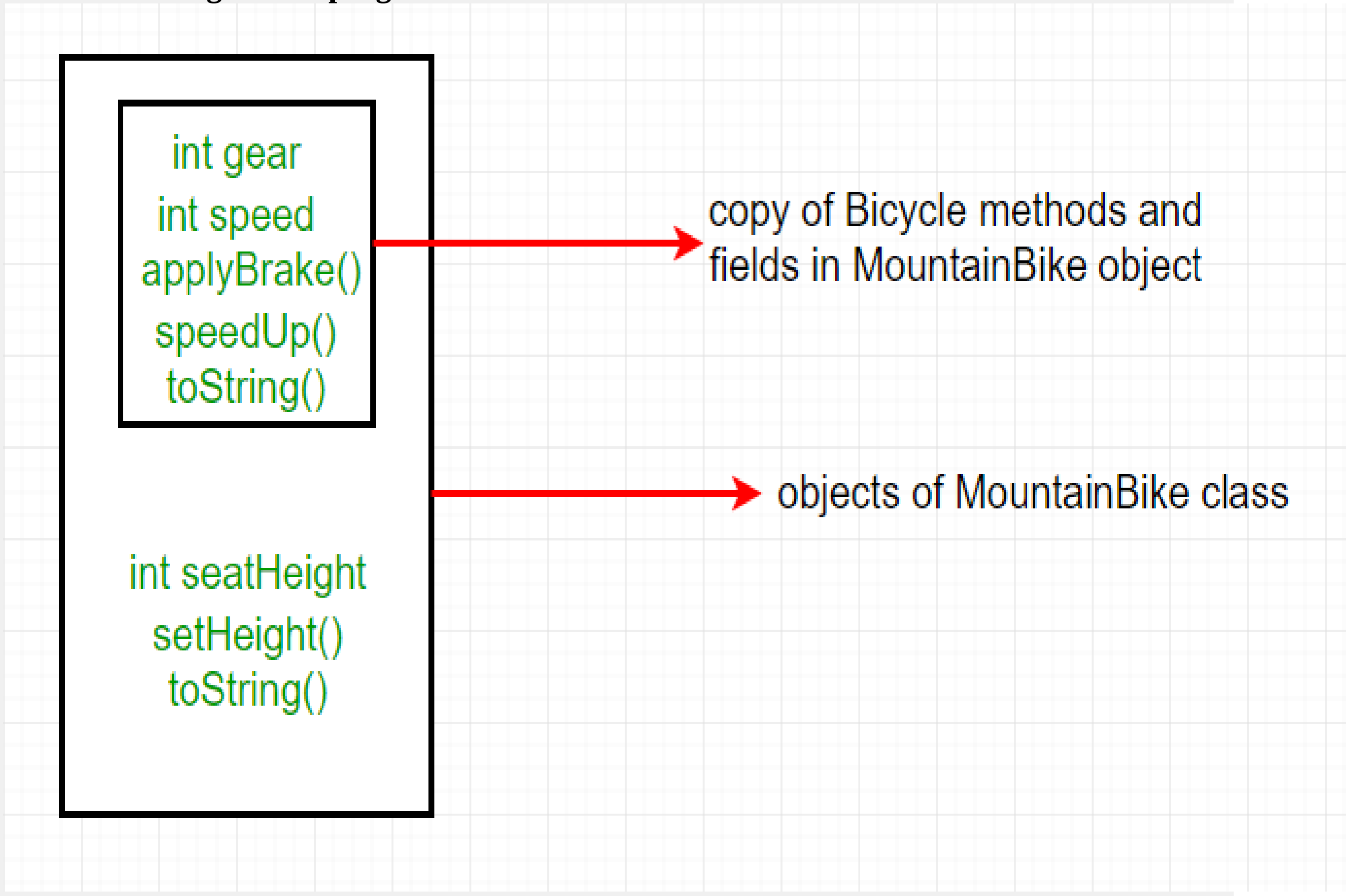
```
// derived class

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. For more, refer Java Object Creation of Inherited Class.

Illustrative image of the program:

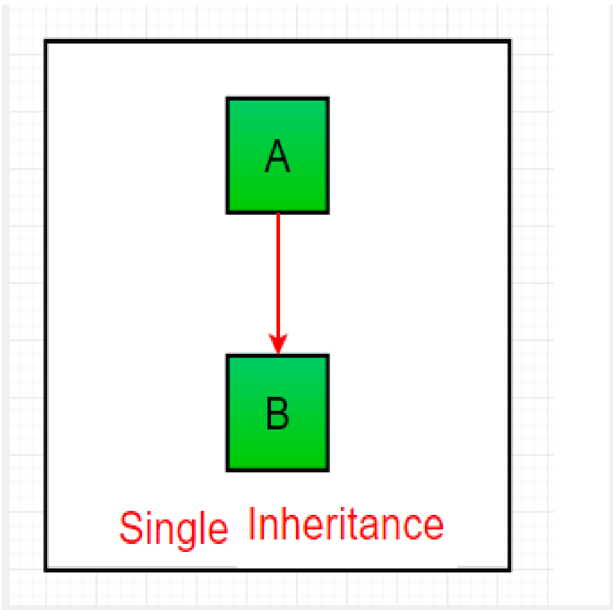

int gear
int speed
applyBrake()
speedUp()
toString()
copy of Bicycle methods and fields in MountainBike object
objects of MountainBike class
int seatHeight
setHeight()
toString()

In practice, inheritance and polymorphism are used together in java to achieve fast performance and readability of code.

Types of Inheritance in Java

Below are the different types of inheritance which is supported by Java.

1. **Single Inheritance :** In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.



filter_none
edit
play_arrow
brightness_4

```
//Java program to illustrate the

// concept of single inheritance

import java.util.*;

import java.lang.*;

import java.io.*;



class one
```

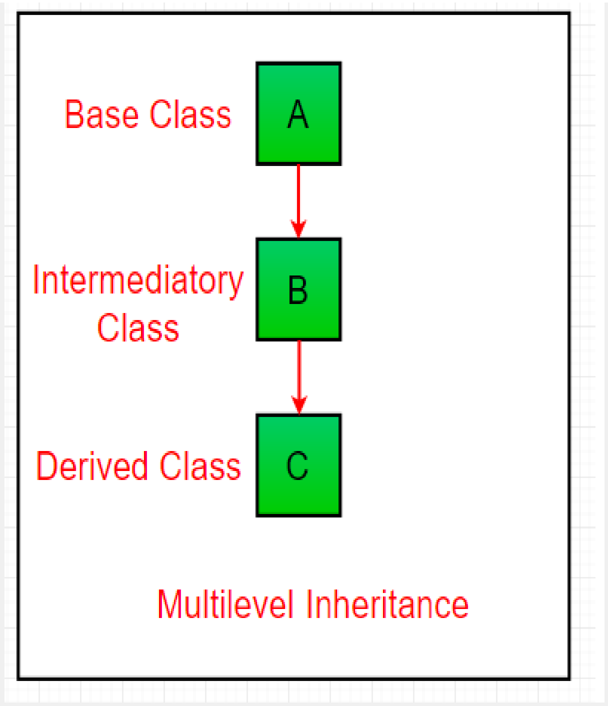
```
{

    public void print_geek()

    {

        System.out.println("Geeks");

    }

}



class two extends one

{

    public void print_for()

    {

        System.out.println("for");

    }

}

// Driver class

public class Main
```

```
{
    public static void main(String[] args)
    {
        two g = new two();
        g.print_geek();
        g.print_for();
        g.print_geek();
    }
}
```

Output:

```
Geeks
for
Geeks
```

2. **Multilevel Inheritance :** In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, 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. In Java, a class cannot directly access the grandparent's members.



filter_none
edit
play_arrow
brightness_4

```
// Java program to illustrate the

// concept of Multilevel inheritance

import java.util.*;

import java.lang.*;

import java.io.*;
```

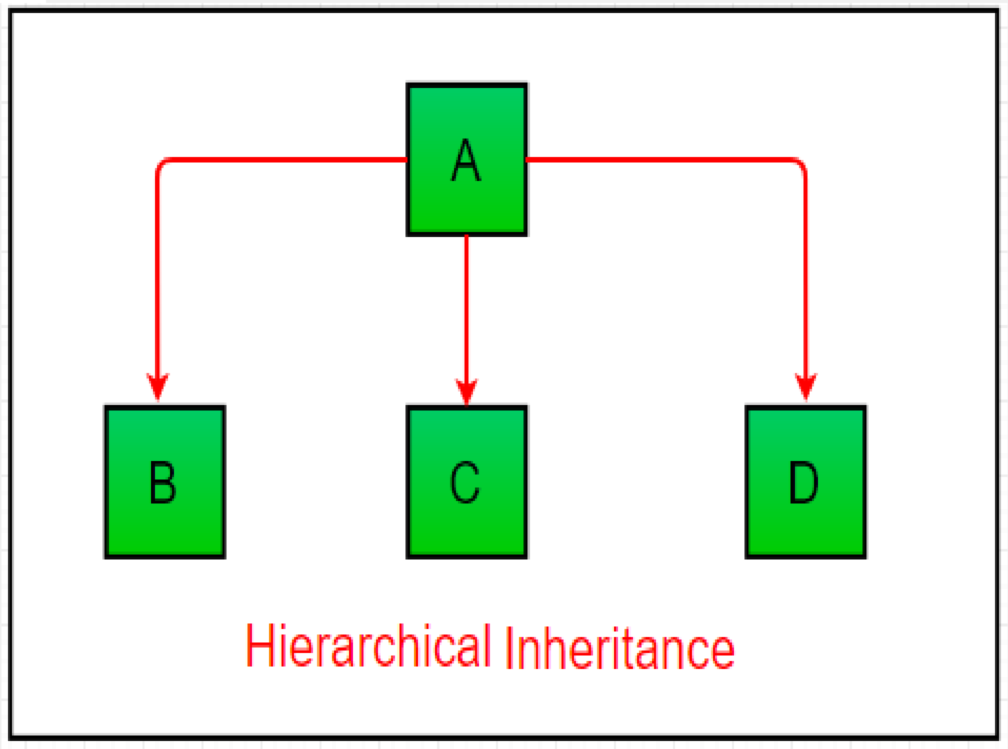
```
class one

{

    public void print_geek()

    {

        System.out.println("Geeks");

    }

}



class two extends one

{

    public void print_for()

    {

        System.out.println("for");

    }

}
```

```
class three extends two

{

    public void print_geek()

    {

        System.out.println("Geeks");

    }

}



// Drived class

public class Main

{

    public static void main(String[] args)

    {

        three g = new three();
        g.print_geek();

        g.print_for();

        g.print_geek();

    }

}
```

Output:

```
Geeks
for
Geeks
```

3. **Hierarchical Inheritance :** In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class.In below image, the class A serves as a base class for the derived class B,C and D.



filter_none

edit
play_arrow
brightness_4

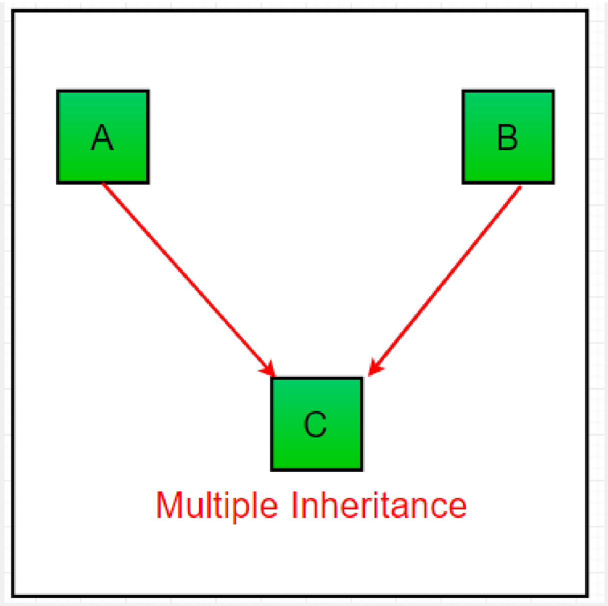
```
// Java program to illustrate the

// concept of Hierarchical inheritance

import java.util.*;

import java.lang.*;

import java.io.*;

  class one

{

    public void print_geek()

    {

        System.out.println("Geeks");

    }

}



class two extends one
```

```
{

    public void print_for()

    {

        System.out.println("for");

    }

}



class three  extends one

{

    /*............*/

}



// Drived class

public class Main

{
    public static void main(String[] args)

    {

        three g = new three();

        g.print_geek();

        two t = new two();

        t.print_for();

        g.print_geek();

    }

}
```

Output:

```
Geeks

for

Geeks
```

4. **Multiple Inheritance (Through Interfaces) :** In Multiple inheritance ,one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support multiple inheritance with classes. In java, we can achieve multiple inheritance only through Interfaces. In image below, Class C is derived from interface A and B.



filter_none
edit
play_arrow
brightness_4

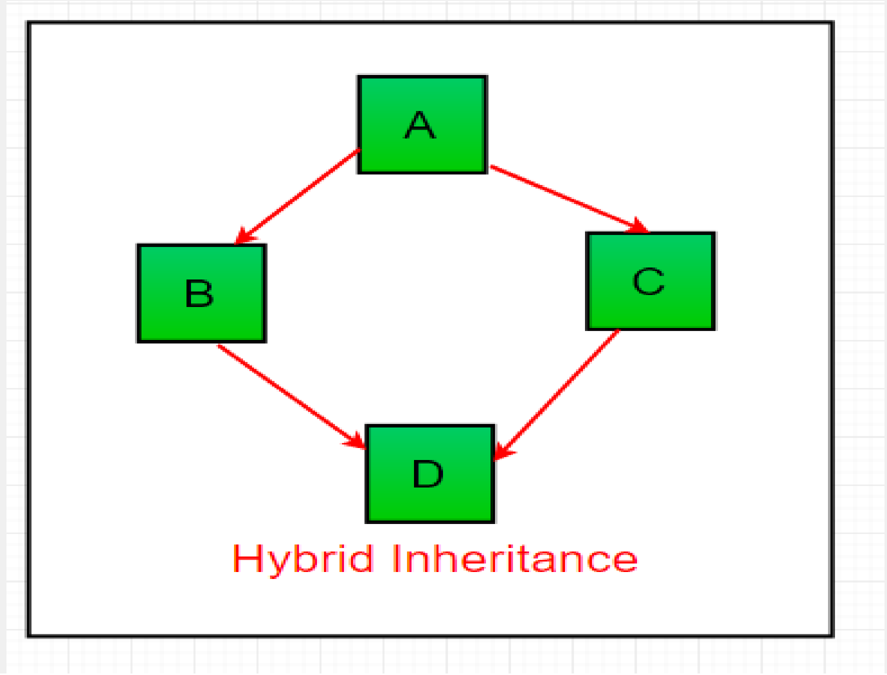
```
// Java program to illustrate the

// concept of Multiple inheritance

import java.util.*;

import java.lang.*;

import java.io.*;



interface one
```

```
{

    public void print_geek();

}



interface two

{

    public void print_for();

}



interface three  extends one,two

{

    public void print_geek();

}

class child  implements three

{
    @Override

    public void print_geek() {

        System.out.println("Geeks");

    }

       public void print_for()

    {

        System.out.println("for");

    }

}

 // Drived class

public class Main

{

    public static void main(String[] args)

    {

        child c = new child();

        c.print_geek();
        c.print_for();


        c.print_geek();


    }


}
```

Output:

```
Geeks
for
Geeks
```

5. **Hybrid Inheritance(Through Interfaces) :** It is a mix of two or more of the above types of inheritance. Since java doesn't support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through Interfaces.



Important facts about inheritance in Java

- **Default superclass**: Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object class.

- **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because Java does not support multiple inheritance with classes. Although with interfaces, multiple inheritance is supported by java.
- **Inheriting Constructors:** A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
- **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods(like getters and setters) for accessing its private fields, these can also be used by the subclass.

What all can be done in a Subclass?

In sub-classes we can inherit members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.
- We can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- We can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus overriding it (as in example above, *toString()* method is overridden).
- We can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus hiding it.
- We can declare new methods in the subclass that are not in the superclass.
- We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

Chapter-5

Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

Example

Let us look at an example.

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above examples −

- A Deer IS-A Animal
- A Deer IS-A Vegetarian
- A Deer IS-A Deer
- A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal −

Example

```
Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

All the reference variables d, a, v, o refer to the same Deer object in the heap.

2nd Definition

Polymorphism in Java

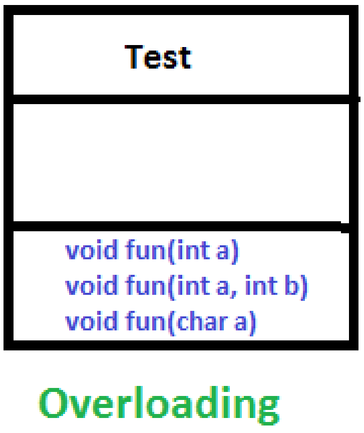
The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

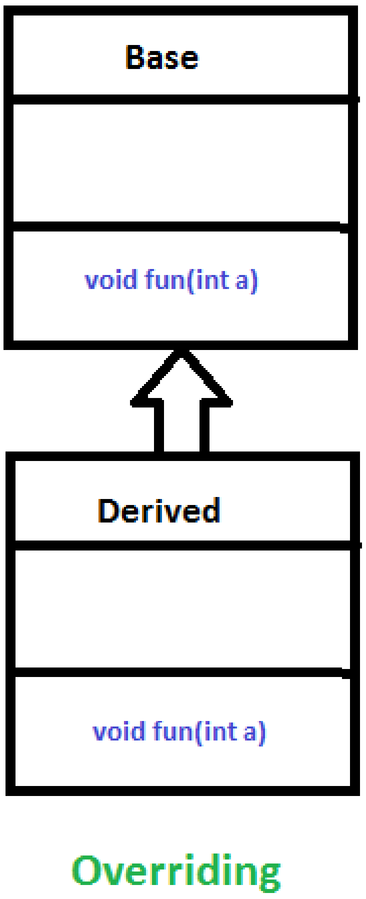
Real life example of polymorphism: A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.
Polymorphism is considered as one of the 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. The word "poly" means many and "morphs" means forms, So it means many forms.

n Java polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism

1. **Compile time polymorphism**: It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.





- **Method Overloading**: When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.
  1. Example: By using different types of arguments

     filter_none
     edit
     play_arrow
     brightness_4

```
// Java program for Method overloading

class MultiplyFun {

    // Method with 2 parameter
    static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Method with the same name but 2 double parameter
    static double Multiply(double a, double b)
    {
        return a * b;
    }
}

class Main {
    public static void main(String[] args)
    {

        System.out.println(MultiplyFun.Multiply(2, 4));

        System.out.println(MultiplyFun.Multiply(5.5, 6.3));
    }
}
```

     Output:

```
8

34.65
```

  2. **Example:** By using different numbers of arguments

     filter_none
     edit
     play_arrow
     brightness_4

```
// Java program for Method overloading

class MultiplyFun {

    // Method with 2 parameter
    static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Method with the same name but 3 parameter
    static int Multiply(int a, int b, int c)
    {
        return a * b * c;
    }
}

class Main {
    public static void main(String[] args)
    {
        System.out.println(MultiplyFun.Multiply(2, 4));

        System.out.println(MultiplyFun.Multiply(2, 7, 3));
    }
}
```

Output:

```
8

42
```

- **Operator Overloading**: Java also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator '+' when placed between integer operands, adds them and when placed between string operands, concatenates them.
  In java, Only "+" operator can be overloaded:
  - To add integers
  - To concatenate strings

  Example:

```
// Java program for Operator overloading

class OperatorOVERDDN {

    void operator(String str1, String str2)
    {
        String s = str1 + str2;
        System.out.println("Concatinated String - "
                           + s);
    }

    void operator(int a, int b)
    {
        int c = a + b;
        System.out.println("Sum = " + c);
    }
}

class Main {
    public static void main(String[] args)
    {
        OperatorOVERDDN obj = new OperatorOVERDDN();
        obj.operator(2, 3);
        obj.operator("joe", "now");
    }
}
```

Output:

```
Sum = 5

Concatinated String - joenow
```

2. **Runtime polymorphism**: It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.
   - **Method overriding**, on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.
     Example:

```
// Java program for Method overridding

class Parent {

    void Print()
    {
        System.out.println("parent class");
    }
}

class subclass1 extends Parent {

    void Print()
    {
        System.out.println("subclass1");
    }
}

class subclass2 extends Parent {

    void Print()
    {
        System.out.println("subclass2");
    }
}

class TestPolymorphism3 {
    public static void main(String[] args)
    {

        Parent a;
```

```
            a = new subclass1();
            a.Print();

            a = new subclass2();
            a.Print();
        }
    }
```

Output:

```
subclass1

subclass2
```

METHOD & CONSTRUCTOR OVERLOADING

Method Overloading

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to constructor overloading in Java, that allows a class to have more than one constructor having different argument lists.

let’s get back to the point, when I say argument list it means the parameters that a method has: For example the argument list of a method add(int a, int b) having two parameters is different from the argument list of the method add(int a, int b, int c) having three parameters.

Three ways to overload a method

In order to overload a method, the argument lists of the methods must differ in either of these:
1. Number of parameters.
For example: This is a valid case of overloading

```
add(int, int)
add(int, int, int)
```

2. Data type of parameters.
For example:

```
add(int, int)
add(int, float)
```
3. Sequence of Data type of parameters.
For example:

```
add(int, float)
add(float, int)
```
Invalid case of method overloading:
When I say argument list, I am not talking about return type of the method, for example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

```
int add(int, int)
float add(int, int)
```
Method overloading is an example of Static Polymorphism. We will discuss polymorphism and types of it in a separate tutorial.

Points to Note:
1. Static Polymorphism is also known as compile time binding or early binding.
2. Static binding happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

Method Overloading examples

As discussed in the beginning of this guide, method overloading is done by declaring same method with different parameters. The parameters must be different in either of these: number, sequence or types of parameters (or arguments). Lets see examples of each of these cases.

Argument list is also known as parameter list

Example 1: Overloading – Different Number of parameters in argument list

This example shows how method overloading is done by having different number of parameters

```
class DisplayOverloading
{
    public void disp(char c)
    {
         System.out.println(c);
    }
    public void disp(char c, int num)
    {
         System.out.println(c + " "+num);
    }
}
class Sample
{
   public static void main(String args[])
   {
       DisplayOverloading obj = new DisplayOverloading();
       obj.disp('a');
       obj.disp('a',10);
   }
}
```
Output:

```
a
a 10
```
In the above example – method disp() is overloaded based on the number of parameters – We have two methods with the name disp but the parameters they have are different. Both are having different number of parameters.

Example 2: Overloading – Difference in data type of parameters

In this example, method disp() is overloaded based on the data type of parameters – We have two methods with the name disp(), one with parameter of char type and another method with the parameter of int type.

```
class DisplayOverloading2
{
    public void disp(char c)
    {
        System.out.println(c);
    }
    public void disp(int c)
    {
       System.out.println(c );
    }
}

class Sample2
{
    public static void main(String args[])
    {
        DisplayOverloading2 obj = new DisplayOverloading2();
        obj.disp('a');
        obj.disp(5);
    }
}
```
Output:

```
a
5
```

Example3: Overloading – Sequence of data type of arguments

Here method disp() is overloaded based on sequence of data type of parameters – Both the methods have different sequence of data type in argument list. First method is having argument list as (char, int) and second is having (int, char). Since the sequence is different, the method can be overloaded without any issues.

```
class DisplayOverloading3
{
   public void disp(char c, int num)
   {
       System.out.println("I’m the first definition of method disp");
   }
   public void disp(int num, char c)
   {
       System.out.println("I’m the second definition of method disp" );
   }
}
class Sample3
{
   public static void main(String args[])
   {
       DisplayOverloading3 obj = new DisplayOverloading3();
       obj.disp('x', 51 );
       obj.disp(52, 'y');
   }
}
```

Output:

```
I’m the first definition of method disp
I’m the second definition of method disp
```

Method Overloading and Type Promotion

When a data type of smaller size is promoted to the data type of bigger size than this is called type promotion, for example: byte data type can be promoted to short, a short data type can be promoted to int, long, double etc.

What it has to do with method overloading?
Well, it is very important to understand type promotion else you will think that the program will throw compilation error but in fact that program will run fine because of type promotion.
Lets take an example to see what I am talking here:

```
class Demo{
   void disp(int a, double b){
        System.out.println("Method A");
   }
   void disp(int a, double b, double c){
        System.out.println("Method B");
   }
   public static void main(String args[]){
        Demo obj = new Demo();
        /* I am passing float value as a second argument but
         * it got promoted to the type double, because there
         * wasn't any method having arg list as (int, float)
         */
        obj.disp(100, 20.67f);
   }
}
```

Output:

```
Method A
```

As you can see that I have passed the float value while calling the disp() method but it got promoted to the double type as there wasn’t any method with argument list as (int, float)

But this type promotion doesn’t always happen, lets see another example:

```
class Demo{
   void disp(int a, double b){
        System.out.println("Method A");
   }
   void disp(int a, double b, double c){
        System.out.println("Method B");
   }
   void disp(int a, float b){
        System.out.println("Method C");
   }
   public static void main(String args[]){
        Demo obj = new Demo();
        /* This time promotion won't happen as there is
         * a method with arg list as (int, float)
         */
        obj.disp(100, 20.67f);
   }
}
```

Output:

```
Method C
```

As you see that this time type promotion didn’t happen because there was a method with matching argument type.

Type Promotion table:

The data type on the left side can be promoted to the any of the data type present in the right side of it.

```
byte → short → int → long
short → int → long
int → long → float → double
float → double
long → float → double
```

Lets see few Valid/invalid cases of method overloading

Case 1:

```
int mymethod(int a, int b, float c)
int mymethod(int var1, int var2, float var3)
```

Result: Compile time error. Argument lists are exactly same. Both methods are having same number, data types and same sequence of data types.

Case 2:

```
int mymethod(int a, int b)
int mymethod(float var1, float var2)
```

Result: Perfectly fine. Valid case of overloading. Here data types of arguments are different.

Case 3:

```
int mymethod(int a, int b)
int mymethod(int num)
```

Result: Perfectly fine. Valid case of overloading. Here number of arguments are different.

Case 4:

```
float mymethod(int a, float b)
float mymethod(float var1, int var2)
```

Result: Perfectly fine. Valid case of overloading. Sequence of the data types of parameters are different, first method is having (int, float) and second is having (float, int).

Case 5:

```
int mymethod(int a, int b)
float mymethod(int var1, int var2)
```

Result: Compile time error. Argument lists are exactly same. Even though return type of methods are different, it is not a valid case. Since return type of method doesn’t matter while overloading a method.

Constructor Overloading

What is a Constructor?

A constructor is a block of code used to create object of a class. Every class has a constructor, be it normal class or abstract class. A constructor is just like a method but without return type. When there is no any constructor defined for a class, a default constructor is created by compiler.

Sometimes there is a need of initializing an object in different ways. This can be done using constructor overloading. For example, Thread class has 8 types of constructors. If we do not want to specify anything about a thread then we can simply use default constructor of Thread class, however if we need to specify thread name, then we may call the parameterized constructor of Thread class with a String args like this:

```
Thread t= new Thread (" MyThread ");
```

Let us take an example to understand need of constructor overloading. Consider the following implementation of a class Box with only one constructor taking three arguments.

```
// An example class to understand need of
// constructor overloading.
class Box
{
    double width, height,depth;

    // constructor used when all dimensions
    // specified
    Box(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }

    // compute and return volume
    double volume()
    {
        return width * height * depth;
    }
}
```

As we can see that the Box() constructor requires three parameters. This means that all declarations of Box objects must pass three arguments to the Box() constructor. For example, the following statement is currently invalid:

```
Box ob = new Box();
```

Since Box() requires three arguments, it’s an error to call it without them. Suppose we simply wanted a box object without initial dimension, or want to initialize a cube by specifying only one value that would be used for all three dimensions. From the above implementation of Box class these options are not available to us.

These types of problems of different ways of initializing an object can be solved by constructor overloading. Below is the improved version of class Box with constructor overloading.

```
// Java program to illustrate
// Constructor Overloading
class Box
{
    double width, height, depth;

    // constructor used when all dimensions
    // specified
    Box(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions
    // specified
    Box()
    {
        width = height = depth = 0;
    }

    // constructor used when cube is created
    Box(double len)
    {
        width = height = depth = len;
    }

    // compute and return volume
    double volume()
    {
        return width * height * depth;
    }
}

// Driver code
public class Test
{
    public static void main(String args[])
    {
        // create boxes using the various
        // constructors
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);

        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println(" Volume of mybox1 is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println(" Volume of mybox2 is " + vol);

        // get volume of cube
        vol = mycube.volume();
        System.out.println(" Volume of mycube is " + vol);
    }
}
```

Output:

```
Volume of mybox1 is 3000.0

Volume of mybox2 is 0.0

Volume of mycube is 343.0
```

Using this() in constructor overloading

this() reference can be used during constructor overloading to call default constructor implicitly from parameterized constructor. Please note, this() should be the first statement inside a constructor.

```
// Java program to illustrate role of this() in
// Constructor Overloading
class Box
{
    double width, height, depth;
    int boxNo;

    // constructor used when all dimensions and
    // boxNo specified
    Box(double w, double h, double d, int num)
    {
        width = w;
        height = h;
        depth = d;
        boxNo = num;
    }

    // constructor used when no dimensions specified
    Box()
    {
        // an empty box
        width = height = depth = 0;
    }

    // constructor used when only boxNo specified
    Box(int num)
    {
        // this() is used for calling the default
        // constructor from parameterized constructor
        this();

        boxNo = num;
    }

    public static void main(String[] args)
    {
        // create box using only boxNo
        Box box1 = new Box(1);

        // getting initial width of box1
        System.out.println(box1.width);
    }
}
```

Output:

```
0.0
```

As we can see in the above program that we called Box(int num) constructor during object creation using only box number. By using this() statement inside it, the default constructor(Box()) is implicitly called from it which will initialize dimension of Box with 0.

Note : The constructor calling should be first statement in the constructor body. For example, following fragment is invalid and throws compile time error.

```
Box(int num)

{

    boxNo = num;


    /* Constructor call must be the first

       statement in a constructor */

    this();  /*ERROR*/

}
```

Important points to be taken care while doing Constructor Overloading :

- Constructor calling must be the **first** statement of constructor in Java.
- If we have defined any parameterized constructor, then compiler will not create default constructor. and vice versa if we don't define any constructor, the compiler creates the default constructor(also known as no-arg constructor) by default during compilation
- Recursive constructor calling is invalid in java.

Chapter-7

Exception Handling

What Is an Exception?

The term exception is shorthand for the phrase "exceptional event."

Definition: An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

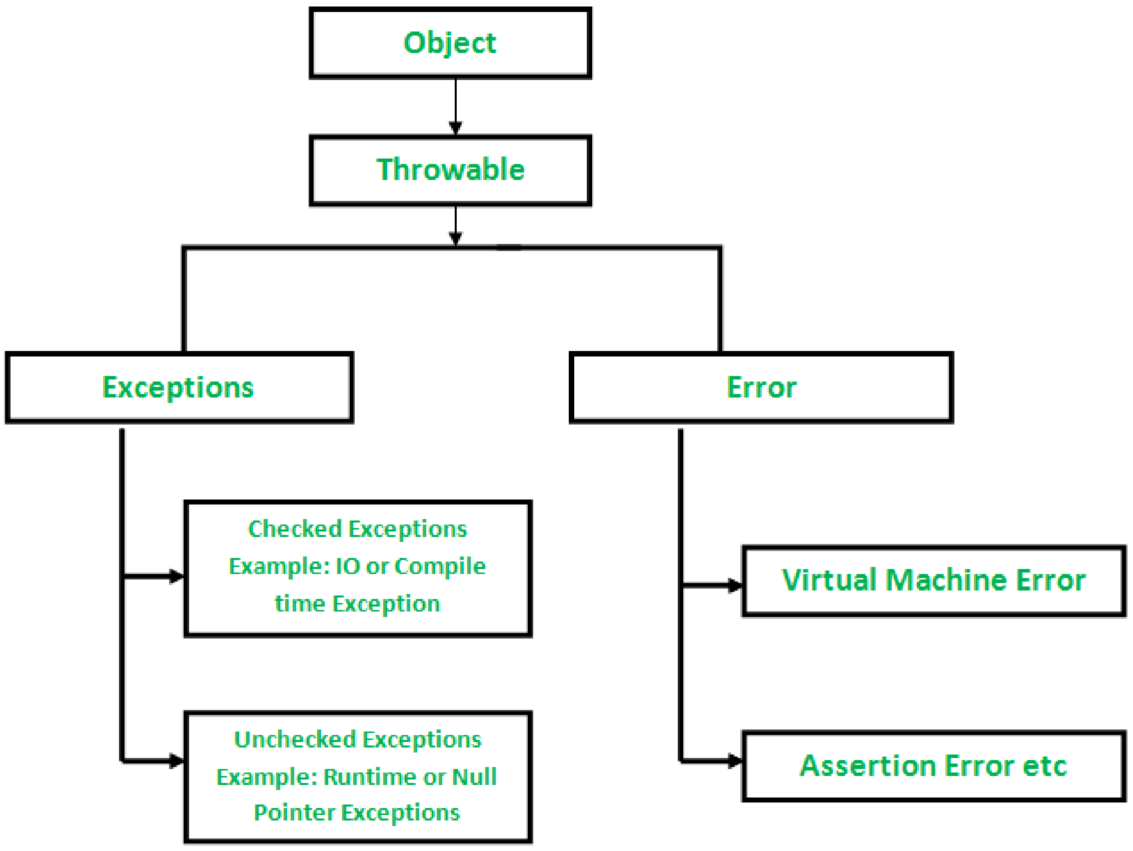
What is an Exception?

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program’s instructions.

Error vs Exception
Error: An Error indicates serious problem that a reasonable application should not try to catch.
Exception: Exception indicates conditions that a reasonable application might try to catch.

Exception Hierarchy

All exception and errors types are sub classes of class **Throwable**, which is base class of hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, **Error** are used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.


Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

How JVM handle an Exception?

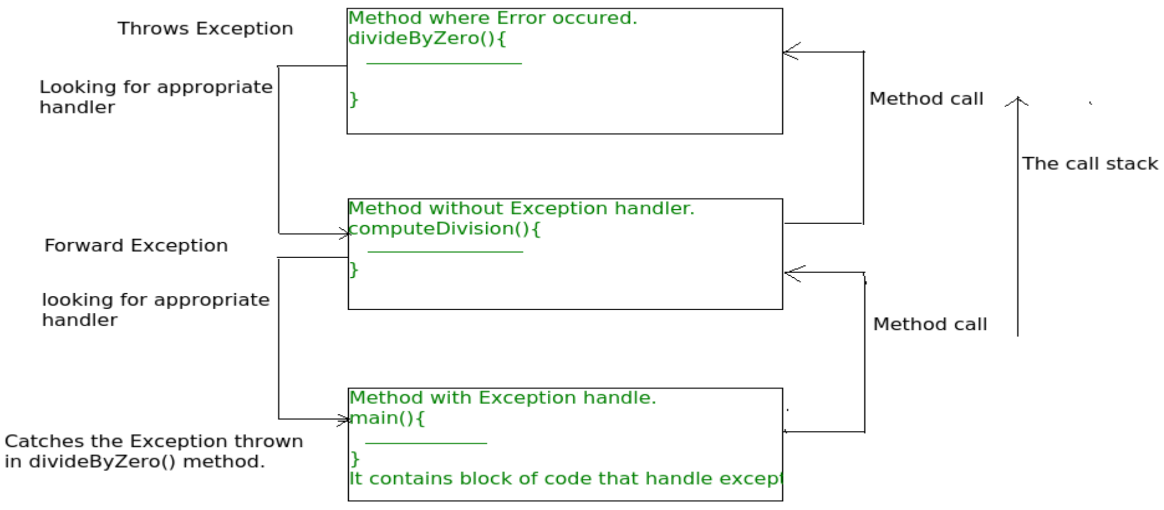
Default Exception Handling: Whenever inside a method, if an exception has occurred, the method creates an Object known as Exception Object and hands it off to the run-time system (JVM). The exception object contains name and description of the exception, and current state of the program where exception has occurred. Creating the Exception Object and handling it to the run-time system is called throwing an Exception. There might be the list of the methods that had been called to get to the method where exception was occurred. This ordered list of the methods is called **Call Stack**. Now the following procedure will happen.

- The run-time system searches the call stack to find the method that contains block of code that can handle the occurred exception. The block of the code is called **Exception handler**.
- The run-time system starts searching from the method in which exception occurred, proceeds through call stack in the reverse order in which methods were called.

- If it finds appropriate handler then it passes the occurred exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
- If run-time system searches all the methods on call stack and couldn't have found the appropriate handler then run-time system handover the Exception Object to **default exception handle**, which is part of run-time system. This handler prints the exception information in the following format and terminates program **abnormally**.

```
•  Exception in thread "xxx" Name of Exception: Description

•  ... ...... ..  // Call Stack
```

See the below diagram to understand the flow of the call stack.



Throws Exception
Method where Error occured.
divideByZero(){
}
Looking for appropriate handler
Method call
The call stack
Method without Exception handler.
computeDivision(){
}
Forward Exception
looking for appropriate handler
Method call
Method with Exception handle.
main(){
}
It contains block of code that handle excep
Catches the Exception thrown in divideByZero() method.

The call stack and searching the call stack for exception handler.

Example :

```
// Java program to demonstrate how exception is thrown.
class ThrowsExecp{

    public static void main(String args[]){

        String str = null;
        System.out.println(str.length());

    }
}
```

Output :

```
Exception in thread "main" java.lang.NullPointerException
    at ThrowsExecp.main(File.java:8)
```

Try, catch, throw and throws in Java

What is an Exception?
An exception is an "unwanted or unexpected event", which occurs during the execution of the program i.e, at run-time, that disrupts the normal flow of the program's instructions. When an exception occurs, execution of the program gets terminated.

Why does an Exception occurs?
An exception can occur due to several reasons like Network connection problem, Bad input provided by user, Opening a non-existing file in your program etc

Blocks & Keywords used for exception handling

1.**try**: The try block contains set of statements where an exception can occur.

```
try

{

    // statement(s) that might cause exception

}
```

2.**catch** : Catch block is used to handle the uncertain condition of try block. A try block is always followed by a catch block, which handles the exception that occurs in associated try block.

```
catch

{

   // statement(s) that handle an exception

   // examples, closing a connection, closing

   // file, exiting the process after writing

   // details to a log file.

}
```

3.**throw**: Throw keyword is used to transfer control from try block to catch block.

4.**throws**: Throws keyword is used for exception handling without try & catch block. It specifies the exceptions that a method can throw to the caller and does not handle itself.

5.**finally**: It is executed after catch block. We basically use it to put some common code when there are multiple catch blocks.

Example of an Exception generated by system is given below :

```
Exception in thread "main"

java.lang.ArithmeticException: divide
by zero at ExceptionDemo.main(ExceptionDemo.java:5)
ExceptionDemo: The class name
main:The method name
ExceptionDemo.java:The file name
java:5:line number
```

```
// Java program to demonstrate working of try,

// catch and finally


class Division {

    public static void main(String[] args)

    {

        int a = 10, b = 5, c = 5, result;

        try {

            result = a / (b - c);

            System.out.println("result" + result);

        }


        catch (ArithmeticException e) {

            System.out.println("Exception caught:Division by zero");

        }


        finally {

            System.out.println("I am in final block");

        }

    }

}
```

Output:

```
Exception caught:Division by zero

I am in final block
```

An example of throws keyword:

```
// Java program to demonstrate working of throws
```

```java
class ThrowsExecp {

    // This method throws an exception
    // to be handled
    // by caller or caller
    // of caller and so on.
    static void fun() throws IllegalAccessException
    {
        System.out.println("Inside fun(). ");
        throw new IllegalAccessException("demo");
    }

    // This is a caller function
    public static void main(String args[])
    {
        try {
            fun();
        }
        catch (IllegalAccessException e) {
            System.out.println("caught in main.");
        }
    }
}
```

Output:

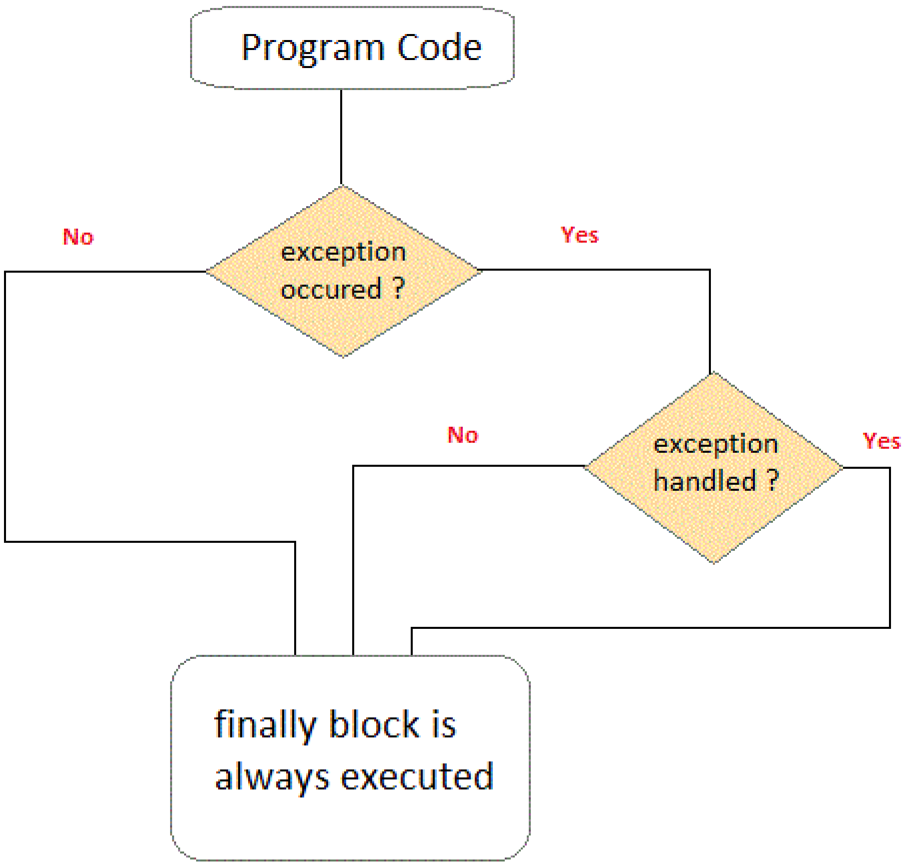
```
Inside fun().
caught in main.
```

Difference between throw and throws

throw	**throws**
throw keyword is used to throw an exception explicitly.	throws keyword is used to declare an exception possible during its execution.
throw keyword is followed by an instance of Throwable class or one of its sub-classes.	throws keyword is followed by one or more Exception class names separated by commas.
throw keyword is declared inside a method body.	throws keyword is used with method signature (method declaration).
We cannot throw multiple exceptions using throw keyword.	We can declare multiple exceptions (separated by commas) using throws keyword.

`finally` clause

A finally keyword is used to create a block of code that follows a try block. A finally block of code is always executed whether an exception has occurred or not. Using a finally block, it lets you run any cleanup type statements that you want to execute, no matter what happens in the protected code. A finally block appears at the end of catch block.



Example demonstrating finally Clause

```
Class ExceptionTest

{

  public static void main(String[] args)

  {

    int a[] = new int[2];

    System.out.println("out of try");

    try

    {

      System.out.println("Access invalid element"+ a[3]);

      /* the above statement will throw ArrayIndexOutOfBoundException */

    }

    finally

    {

      System.out.println("finally is always executed.");
    }
  }
}
```

```
Out of try

finally is always executed.

Exception in thread main java. Lang. exception array Index out of bound exception.
```

User defined Exception subclass in Java

You can also create your own exception sub class simply by extending java **Exception** class. You can define a constructor for your Exception sub class (not compulsory) and you can override the **toString()** function to display your customized message on catch.

```
class MyException extends Exception
{
  private int ex;
  MyException(int a)
  {
    ex = a;
  }
  public String toString()
  {
    return "MyException[" + ex +"] is less than zero";
  }
}


class Test
{
  static void sum(int a,int b) throws MyException
  {
    if(a<0)
    {
      throw new MyException(a); //calling constructor of user-defined exception
class
    }
    else
    {
      System.out.println(a+b);
    }
  }

  public static void main(String[] args)
  {
    try
    {
      sum(-10, 10);
    }
    catch(MyException me)
    {
      System.out.println(me); //it calls the toString() method of user-defined
Exception
    }
  }
}
```

```
MyException[-10] is less than zero
```

Importance of exception handling in practical implementation of live projects.

Many kinds of errors can cause exceptions--problems ranging from serious hardware errors, such as a hard disk crash, to simple programming errors, such as trying to access an out-of-bounds array element. When such an error occurs within a Java method, the method creates an exception object and hands it off to the runtime system. The exception object contains information about the exception, including its type and the state of the program when the error occurred. The runtime system is then responsible for finding some code to handle the error. In Java terminology, creating an exception object and handing it to the runtime system is called *throwing an exception*.

After a method throws an exception, the runtime system leaps into action to find someone to handle the exception. The set of possible "someones" to handle the exception is the set of methods in the call stack of the method where the error occurred. The runtime system searches backwards through the call stack, beginning with the method in which the error occurred, until it finds a method that contains an appropriate *exception handler*. An exception handler is considered appropriate if the type of the exception thrown is the same as the type of exception handled by the handler. Thus the exception bubbles up through the call stack until an appropriate handler is found and one of the calling methods handles the exception. The exception handler chosen is said to *catch the exception*.

If the runtime system exhaustively searches all of the methods on the call stack without finding an appropriate exception handler, the runtime system (and consequently the Java program) terminates.

By using exceptions to manage errors, Java programs have the following advantages over traditional error management techniques:

- Advantage 1: Separating Error Handling Code from "Regular" Code
- Advantage 2: Propagating Errors Up the Call Stack
- Advantage 3: Grouping Error Types and Error Differentiation

Advantage 1: Separating Error Handling Code from "Regular" Code

In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code. For example, suppose that you have a function that reads an entire file into memory. In pseudo-code, your function might look something like this:

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

At first glance this function seems simple enough, but it ignores all of these potential errors:

- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if enough memory can't be allocated?
- What happens if the read fails?
- What happens if the file can't be closed?

To answer these questions within your **`read_file`** function, you'd have to add a lot of code to do error detection, reporting and handling. Your function would end up looking something like this:

```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

With error detection built in, your original 7 lines (in bold) have been inflated to 29 lines of code--a bloat factor of almost 400 percent. Worse, there's so much error detection, reporting, and returning that the original 7 lines of code are lost in the clutter. And worse yet, the logical flow of the code has also been lost in the clutter, making it difficult to tell if the code is doing the right thing: Is the file *really* being closed if the function fails to allocate enough memory? It's even more difficult to ensure that the code continues to do the right thing after you modify the function three months after writing it. Many programmers "solve" this problem by simply ignoring it--errors are "reported" when their programs crash.

Java provides an elegant solution to the problem of error management: exceptions. Exceptions enable you to write the main flow of your code and deal with the, well, exceptional cases elsewhere. If your `read_file` function used exceptions instead of traditional error management techniques, it would look something like this:

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Note that exceptions don't spare you the effort of doing the work of detecting, reporting, and handling errors. What exceptions do provide for you is the means to separate all the grungy details of what to do when something out-of-the-ordinary happens from the main logic of your program.

In addition, the bloat factor for error management code in this program is about 250 percent--compared to 400 percent in the previous example.

Advantage 2: Propagating Errors Up the Call Stack

A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods. Suppose that the `readFile` method is the fourth method in a series of nested method calls made by your main program: `method1` calls `method2`, which calls `method3`, which finally calls `readFile`.

```
method1 {
    call method2;
}
method2 {
    call method3;
}
method3 {
    call readFile;
}
```

Suppose also that `method1` is the only method interested in the errors that occur within `readFile`. Traditional error notification techniques force `method2` and `method3` to propagate the error codes returned by `readFile` up the call stack until the error codes finally reach `method1`--the only method that is interested in them.

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}
errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}
errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```

As you learned earlier, the Java runtime system searches backwards through the call stack to find any methods that are interested in handling a particular exception. A Java method can "duck" any exceptions thrown within it, thereby allowing a method further up the call stack to catch it. Thus only the methods that care about errors have to worry about detecting errors.

```
method1 {
    try {
        call method2;
    } catch (exception) {
        doErrorProcessing;
    }
}
method2 throws exception {
    call method3;
}
method3 throws exception {
    call readFile;
}
```

However, as you can see from the pseudo-code, ducking an exception does require some effort on the part of the "middleman" methods. Any checked exceptions that can be thrown within a method are part of that method's public programming interface and must be specified in the `throws` clause of the method. Thus a method informs its callers about the exceptions that it can throw, so that the callers can intelligently and consciously decide what to do about those exceptions.

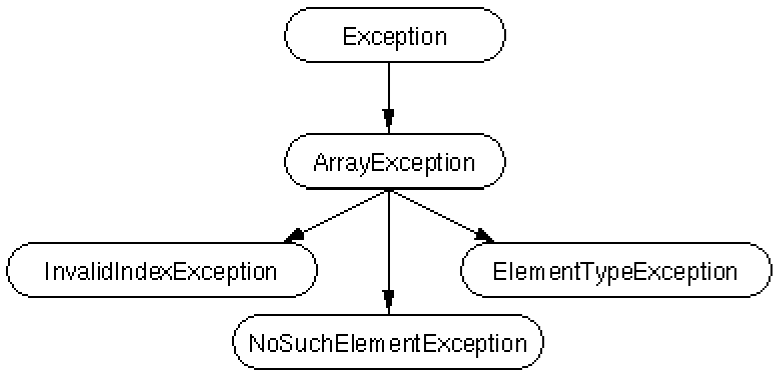
Note again the difference in the bloat factor and code obfuscation factor of these two error management techniques. The code that uses exceptions is more compact and easier to understand.

Advantage 3: Grouping Error Types and Error Differentiation

Often exceptions fall into categories or groups. For example, you could imagine a group of exceptions, each of which represents a specific type of error that can occur when manipulating an array: the index is out of range for the size of the array, the element being inserted into the array is of the wrong type, or the element being searched for is not in the array. Furthermore, you can imagine that some methods would like to handle all exceptions that fall within a category (all array exceptions), and other methods would like to handle specific exceptions (just the invalid index exceptions, please).

Because all exceptions that are thrown within a Java program are first-class objects, grouping or categorization of exceptions is a natural outcome of the class hierarchy. Java exceptions must be instances of `Throwable` or any `Throwable` descendant. As for other Java classes, you can create subclasses of the `Throwable` class and subclasses of your subclasses. Each "leaf" class (a class with no subclasses) represents a specific type of exception and each "node" class (a class with one or more subclasses) represents a group of related exceptions.

For example, in the following diagram, `ArrayException` is a subclass of `Exception` (a subclass of `Throwable`) and has three subclasses.



InvalidIndexException, ElementTypeException, and NoSuchElementException are all leaf classes. Each one represents a specific type of error that can occur when manipulating an array. One way a method can exceptions is to catch only those that are instances of a leaf class. For example, an exception handler that handles only invalid index exceptions has a catch statement like this:

```
catch (InvalidIndexException e) {
    . . .
}
```

ArrayException is a node class and represents any error that can occur when manipulating an array object, including those errors specifically represented by one of its subclasses. A method can catch an exception based on its group or general type by specifying any of the exception's superclasses in the catch statement. For example, to catch all array exceptions regardless of their specific type, an exception handler would specify an ArrayException argument:

```
catch (ArrayException e) {
    . . .
}
```

This handler would catch all array exceptions including InvalidIndexException, ElementTypeException, and NoSuchElementException. You can find out precisely which type of exception occurred by querying the exception handler parameter e. You could even set up an exception handler that handles any Exception with this handler:

```
catch (Exception e) {
    . . .
}
```

Exception handlers that are too general, such as the one shown here, can make your code more error prone by catching and handling exceptions that you didn't anticipate and therefore are not correctly handled within the handler. We don't recommend writing general exception handlers as a rule.

Interface vs Abstract Class in Java

What is Interface?

The interface is a blueprint that can be used to implement a class. The interface does not contain any concrete methods (methods that have code). All the methods of an interface are abstract methods.

An interface cannot be instantiated. However, classes that implement interfaces can be instantiated. Interfaces never contain instance variables but, they can contain public static final variables (i.e., constant class variables)

What Is Abstract Class?

A class which has the abstract keyword in its declaration is called abstract class. Abstract classes should have at least one abstract method. , i.e., methods without a body. It can have multiple concrete methods.

Abstract classes allow you to create blueprints for concrete classes. But the inheriting class should implement the abstract method.

Abstract classes cannot be instantiated.

Important Reasons For Using Interfaces

Interfaces are used to achieve abstraction.

Designed to support dynamic method resolution at run time

It helps you to achieve loose coupling.

Allows you to separate the definition of a method from the inheritance hierarchy

Important Reasons For Using Abstract Class

Abstract classes offer default functionality for the subclasses.

Provides a template for future specific classes

Helps you to define a common interface for its subclasses

Abstract class allows code reusability.

Interface Vs. Abstract Class

arameters	Interface	Abstract class

Speed	Slow	Fast
Multiple Inheritances	Implement several Interfaces	Only one abstract class
Structure	Abstract methods	Abstract & concrete methods
When to use	Future enhancement	To avoid independence
Inheritance/ Implementation	A Class can implement multiple interfaces	The class can inherit only one Abstract Class
Default Implementation	While adding new stuff to the interface, it is a nightmare to find all the implementors and implement newly defined stuff.	In case of Abstract Class, you can take advantage of the default implementation.
Access Modifiers	The interface does not have access modifiers. Everything defined inside the interface is assumed public modifier.	Abstract Class can have an access modifier.
When to use	It is better to use interface when various implementations share only method signature. Polymorphic hierarchy of value types.	It should be used when various implementations of the same kind share a common behavior.
Data fields	the interface cannot contain data fields.	the class can have data fields.
Multiple Inheritance Default	A class may implement numerous interfaces.	A class inherits only one abstract class.
Implementation	An interface is abstract so that it can't provide any code.	An abstract class can give complete, default code which should be overridden.
Use of Access modifiers	You cannot use access modifiers for the method, properties, etc.	You can use an abstract class which contains access modifiers.
Usage	Interfaces help to define the peripheral abilities of a class.	An abstract class defines the identity of a class.

Defined fields	No fields can be defined	An abstract class allows you to define both fields and constants
Inheritance	An interface can inherit multiple interfaces but cannot inherit a class.	An abstract class can inherit a class and multiple interfaces.
Constructor or destructors	An interface cannot declare constructors or destructors.	An abstract class can declare constructors and destructors.
Limit of Extensions	It can extend any number of interfaces.	It can extend only one class or one abstract class at a time.
Abstract keyword	In an abstract interface keyword, is optional for declaring a method as an abstract.	In an abstract class, the abstract keyword is compulsory for declaring a method as an abstract.
Class type	An interface can have only public abstract methods.	An abstract class has protected and public abstract methods.

Sample code for Interface and Abstract Class in Java

Following is sample code to create an interface and abstract class in Java

Interface Syntax

```
interface name{

//methods

}
```

Java Interface Example:

```
interface Pet {

   public void test();

}

class Dog implements Pet {

   public void test() {

      System.out.println("Interface Method Implemented");
```

```
    }
    public static void main(String args[]) {
        Pet p = new Dog();
        p.test();
    }
}
```

Abstract Class Syntax

```
abstract class name{
    // code
}
```

Abstract class example:

```
abstract class Shape {
    int b = 20;
    abstract public void calculateArea();
}

public class Rectangle extends Shape {
    public static void main(String args[]) {
        Rectangle obj = new Rectangle();
        obj.b = 200;
        obj.calculateArea();
    }
    public void calculateArea() {
        System.out.println("Area is " + (obj.b * obj.b));
    }
}
```

Multiple Inheritance Using Interface

Definition

Inheritance is when an object or class is based on another object or class, using the same implementation specifying implementation to maintain the same behavior. It is a mechanism for code reuse and to allow independent extensions of the original software via public classes and interfaces. The relationships of objects or classes through inheritance give rise to a hierarchy. Multiple Inheritance allows a class to have more than one super class and to inherit features from all parent class. it is achieved using interface.

Syntax

```
public interface A{
    //Do Something
}
public interface B extends A{
    //Do Something
}
public interface C extends A{
    //Do Something
}
```

Multiple Inheritance Using Interface Example Program

```

interface vehicleone{
    int  speed=90;
    public void distance();
}

interface vehicletwo{
    int distance=100;
    public void speed();
}

class Vehicle  implements vehicleone,vehicletwo{
    public void distance(){
             int  distance=speed*100;
             System.out.println("distance travelled is "+distance);
    }
    public void speed(){
             int speed=distance/100;
    }
}

class MultipleInheritanceUsingInterface{
    public static void main(String args[]){
             System.out.println("Vehicle");
             obj.distance();
             obj.speed();
    }
}
```

Multiple Inheritance is a feature of object oriented concept, where a class can inherit properties of more than one parent class. The problem occurs when there exist methods with same signature in both the super classes and subclass. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority.

Why Java doesn't support Multiple Inheritance?

Consider the below Java code. It shows error.

filter_none

edit

play_arrow

brightness_4

```
// First Parent class
class Parent1
{
    void fun()
    {
        System.out.println("Parent1");
    }
}

// Second Parent Class
class Parent2
{
    void fun()
    {
        System.out.println("Parent2");
    }
}

// Error : Test is inheriting from multiple
// classes
class Test extends Parent1, Parent2
{
   public static void main(String args[])
   {
       Test t = new Test();
       t.fun();
   }
}
```

Output :

```
Compiler Error
```

From the code, we see that, on calling the method fun() using Test object will cause complications such as whether to call Parent1's fun() or Parent2's fun() method.

1. The Diamond Problem:

```
          GrandParent

           /      \
          /       \
      Parent1      Parent2
          \       /
           \     /
             Test
```

filter_none
edit
play_arrow
brightness_4

```
// A Grand parent class in diamond
class GrandParent
{
    void fun()
    {
        System.out.println("Grandparent");
    }
}

// First Parent class
class Parent1 extends GrandParent
{
    void fun()
    {
        System.out.println("Parent1");
    }
}

// Second Parent Class
class Parent2 extends GrandParent
{
    void fun()
    {
        System.out.println("Parent2");
    }
}



// Error : Test is inheriting from multiple
// classes
class Test extends Parent1, Parent2
{
   public static void main(String args[])
   {
       Test t = new Test();
       t.fun();
   }
}
```

From the code, we see that: On calling the method fun() using Test object will cause complications such as whether to call Parent1’s fun() or Child’s fun() method.

Therefore, in order to avoid such complications Java does not support multiple inheritance of classes.

2. Simplicity – Multiple inheritance is not supported by Java using classes , handling the complexity that causes due to multiple inheritance is very complex. It creates problem during various operations like casting, constructor chaining etc and the above all reason is that there are very few scenarios on which we actually need multiple inheritance, so better to omit it for keeping the things simple and straightforward.

How are above problems handled for Default Methods and Interfaces ?

Java 8 supports default methods where interfaces can provide default implementation of methods. And a class can implement two or more interfaces. In case both the implemented interfaces contain default methods with same method signature, the implementing class should explicitly specify which default method is to be used or it should override the default method.

filter_none
edit
play_arrow
brightness_4

```
// A simple Java program to demonstrate multiple
// inheritance through default methods.
interface PI1
{
    // default method
    default void show()
    {
        System.out.println("Default PI1");
    }
}

interface PI2
{
    // Default method
    default void show()
    {
        System.out.println("Default PI2");
    }
}

// Implementation class code
class TestClass implements PI1, PI2
{
    // Overriding default show method
    public void show()
    {
        // use super keyword to call the show
        // method of PI1 interface
        PI1.super.show();

        // use super keyword to call the show
        // method of PI2 interface
        PI2.super.show();
    }

    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.show();
    }
}
```
Output:

```
Default PI1

Default PI2
```

If we remove implementation of default method from “TestClass”, we get compiler error. See this for a sample run.
If there is a diamond through interfaces, then there is no issue if none of the middle interfaces provide implementation of root interface. If they provide implementation, then implementation can be accessed as above using super keyword.

filter_none
edit
play_arrow
brightness_4

```
// A simple Java program to demonstrate how diamond
// problem is handled in case of default methods

interface GPI
{
    // default method
    default void show()
    {
        System.out.println("Default GPI");
    }
}

interface PI1 extends GPI { }

interface PI2 extends GPI { }

// Implementation class code
class TestClass implements PI1, PI2
{
    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.show();
    }
}
```
Output:

```
Default GPI
```

Chapter -5 (Polymorphism)

Up-casting and down-casting constructor overloading

Perhaps in your daily Java coding, you see (and use) ***upcasting*** and ***down casting*** occasionally. You may hear the terms 'casting', 'upcasting', 'down casting' from someone or somewhere, and you may be confused about them.
As you read on, you will realize that upcasting and down casting are really simple.
Before we go into the details, suppose that we have the following class hierarchy:

`Mammal > Animal > Dog, Cat`

`Mammal` is the super interface:

```
public interface Mammal {

        public void eat();


        public void move();


        public void sleep();

}
```

`Animal` is the abstract class:

```
public abstract class Animal implements Mammal {

        public void eat() {

                System.out.println("Eating...");

        }


        public void move() {

                System.out.println("Moving...");

        }


        public void sleep() {

                System.out.println("Sleeping...");

        }


}
```

`Dog` and `Cat` are the two concrete sub classes:

```
public class Dog extends Animal {

        public void bark() {
```

```
                    System.out.println("Gow gow!");

          }

          public void eat() {

                    System.out.println("Dog is eating...");

          }

}


public class Cat extends Animal {

          public void meow() {

                    System.out.println("Meow Meow!");

          }

}
```

1. What is Upcasting in Java?

Upcasting is casting a subtype to a supertype, upward to the inheritance tree. Let's see an example:

```
Dog dog = new Dog();

Animal anim = (Animal) dog;

anim.eat();
```

Here, we cast the `Dog` type to the `Animal` type. Because Animal is the supertype of `Dog`, this casting is called upcasting.
Note that the actual object type does not change because of casting. The `Dog` object is still a `Dog` object. Only the reference type gets changed. Hence the above code produces the following output:

```
Dog is eating…
```

Upcasting is always safe, as we treat a type to a more general one. In the above example, an `Animal` has all behaviors of a `Dog`.
This is also another example of upcasting:

```
Mammal mam = new Cat();

Animal anim = new Dog();
```

2. Why is Upcasting in Java?

Generally, upcasting is not necessary. However, we need upcasting when we want to write general code that deals with only the supertype. Consider the following class:

```
public class AnimalTrainer {

          public void teach(Animal anim) {

                    anim.move();

                    anim.eat();

          }
}
```

Here, the `teach()` method can accept any object which is subtype of `Animal`. So objects of type `Dog` and Cat will be upcasted to `Animal` when they are passed into this method:

```
Dog dog = new Dog();

Cat cat = new Cat();



AnimalTrainer trainer = new AnimalTrainer();

trainer.teach(dog);

trainer.teach(cat);
```

3. What is Downcasting in Java?

Downcasting is casting to a subtype, downward to the inheritance tree. Let's see an example:

```
Animal anim = new Cat();

Cat cat = (Cat) anim;
```

Here, we cast the `Animal` type to the `Cat` type. As `Cat` is subclass of `Animal`, this casting is called downcasting.
Unlike upcasting, downcasting can fail if the actual object type is not the target object type. For example:

```
Animal anim = new Cat();

Dog dog = (Dog) anim;
```

This will throw a `ClassCastException` because the actual object type is `Cat`. And a `Cat` is not a `Dog` so we cannot cast it to a `Dog`.
The Java language provides the instanceof keyword to check type of an object before casting.
For example:

```
if (anim instanceof Cat) {

         Cat cat = (Cat) anim;

         cat.meow();

} else if (anim instanceof Dog) {

         Dog dog = (Dog) anim;

         dog.bark();

}
```

So if you are not sure about the original object type, use the `instanceof` operator to check the type before casting. This eliminates the risk of a `ClassCastException` thrown.

4. Why is Downcasting in Java?

Downcasting is used more frequently than upcasting. Use downcasting when we want to access specific behaviors of a subtype.
Consider the following example:

```
public class AnimalTrainer {

         public void teach(Animal anim) {

                  // do animal-things
```

```
                    anim.move();

                    anim.eat();


                    // if there's a dog, tell it barks

                    if (anim instanceof Dog) {

                              Dog dog = (Dog) anim;

                              dog.bark();

                    }

          }

}
```

Here, in the `teach()` method, we check if there is an instance of a `Dog` object passed in, downcast it to the `Dog` type and invoke its specific method, `bark()`.
Okay, so far you have got the nuts and bolts of upcasting and downcasting in Java. Remember:

- Casting does not change the actual object type. Only the reference type gets changed.
- Upcasting is always safe and never fails.
- Downcasting can risk throwing a `ClassCastException`, so the `instanceof` operator is used to check type before casting.

2nd Notes

Polymorphism in Java

Meaning of Polymorphism :

- The word **polymorphism** means many-form.
- In OOPs, polymorphism deals with behavior(method or function) part of the object.
- If an object is able to perform single functionality in multiple ways, this is called **polymorphic** behavior of that object.
- Java polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Types of Polymorphism :

On the basis of concept of BINDING, the polymorphism is categorized into two category:

1. Compile Time Polymorphism
2. Runtime Polymorphism

Here, BINDING means the association of function calling with the function definition.

1. Compile Time Polymorphism :

- If BINDING association is known at compile time then this is known as compile time polymorphism.
- This is also known as **static binding** or **early binding**.
- Compile time polymorphism can be achieved in Java programming with the implementation of the following programming concept

Method Overloading

Constructor Overloading

2. Runtime Polymorphism :

- If BINDING association is known at compile time that decides the association at runtime then this is known as runtime polymorphism.
- This is also known as **dynamic binding** or **late binding**.
- Compile time polymorphism can be achieved in Java programming with the implementation of the programming concept

Method Overriding

1. Compile Time Polymorphism :

Method Overloading

- Java allows to create more than one method with the same name in the same class. This mechanism is called **method overloading**.
- In method overloading the methods name are same but the argument list of each method must be different.
- To overload method there are three ways:

1. Define a new method with the **same name** with **different types** of argument.
2. Define a new method with the **same name** with **different number** of argument.
3. Define a new method with the **same name** with **different types** and **different number** of argument.

NOTE: Return type of method, doesn't play any role in method overloading.

Example: Method overloading :

```
class Demo{
 public void show( int a )
  {
      //definition
  }
 //Number of arguments are different
 public void show( int a, int b )
  {
      //definition
  }
 //Types of arguments are different
 public void show( String a, String b )
  {
      //definition
  }

 public void show( int a, String b )
  {
      //definition
  }
 //Order of arguments are different
 public void show( String a, int b )
  {
      //definition
  }
 public static void main(String []arg)
 {
  Demo d = new Demo ();
   d.show(10);
   d.show(10, 20);
   d.show("Java", "Prowess");
   d.show(10, "Java Prowess");
   d.show("Java Prowess", 10);
 }
}
```

Example:

Constructor overloading :

```
class User{
  private String name, role;
  User(String name)
   {
      this.name = name;
   }
  User( String name, String role)
   {
      this.name = name;
      this.role = role;
   }

 public static void main(String []arg)
 {
   User user1 = new User("Ayan");
   User user2 = new User("Atif","DBA");
 }
}
```

1.RunTime Polymorphism :

Method Overriding

- When a method in a **sub-class** has the same name and type signature as a method in its **super-class**, then the method in the sub-class is said to override the method of the super class.
- Both signature and return type must be the same as super class.
- The **throws** clause of an overriding method can have fewer types listed than the method in the super class, or more specific types or both.
- **Example :**

```
//Parent.java
public class Parent
{
 public void show()
  {
   System.out.println("Hello Parent");
  }
}
-------------------------------
//Child.java
public class Child extends Parent
{
//overriding
 public void show()
  {
   System.out.println("Hello Child");
  }
}
```

- A sub-class can change the access specifier of the method of the super-class , but only if it provides more access.
- A method declared **public** in super class, will be public in sub class.
- A method declared **protected** in super class can be re-declared protected or public but not private.

- A method declared **default** in super class can be re-declared default, protected or public but not private.
- Fields cannot be overridden, they can only be hidden.
- To access the hidden fields use the **super** keyword.
- **Note :**
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.
- Only non-static method can be override.
- Final method can't be override.

Example :

```
//Parent.java
public class Parent
{
 public void show()
  {
   System.out.println("show from Parent");
  }
 void display()
  {
   System.out.println("display from Parent");
  }
}
-------------------
//Child.java
public class Child extends Parent
{
 public void show()
  {
   System.out.println("show from Child");
  }
 //more acces
 public void display()
 {
  System.out.println("display from Child");
 }
}
-----------------------
//MainClass.java
public class MainClass
{
 public static void main(String []arg)
  {
    Parent  p = new parent();
        p.show();
        p.display();
    Child ch = new Child();
        ch.show();
        ch.diaplay();
  }
}
```

OUTPUT :

```
show from Parent
display from Parent
show from Child
display from Child
```

Upcasting :

- An upcast is a cast from a derived type to one of its base classes. This cast is safe and does not require an explicit cast notation.
- Upcasting is using the Super class's reference to refer to a Sub class's object. Or we can say that, the act of converting a Sub class's reference into its Super class's reference is called Upcasting.
- ***Syntax :***

```
•  Super-class ref = Child-class Object;
•  e.g.
```

```
Parent p = new Child();
```

- **Note :**
- Through this reference you can access only those methods, which are inherited or override by subclass, child's method can't be access.

Downcasting :

- The process of converting super class's refernce that pointing to sub-class Object, to sub-class reference is called **downcasting**.
- **downcasting** is to be done explicit.
- ***Syntax ;***

```
• Chil-class ref = (Child-class) Parent-ref.
• e.g.
```

```
Child ch = (Child) p;
```

Example :

Program to understand the concept

```
//Parent.java
public class Parent
{
 public void show()
  {
   System.out.println("show from Parent");
  }
 void display()
  {
   System.out.println("display from Parent");
  }
}
-------------------
//Child.java
public class Child extends Parent
{
 //more override
 public void display()
 {
  System.out.println("display from Child");
 }
public void xyz()
 {
  System.out.println("Child's Method");
 }
}
-----------------------
//MainClass.java
public class MainClass
{
 public static void main(String []arg)
  {
    //Upcasting
    Parent p = new Child();
    p.show(); //call inherited method
    p.diaplay();//call override method
    p.xyz(); //ERROR

    //Downcasting
    Child ch = (Child) P;
    ch.show(); //call inherited method
    ch.diaplay();//call override method
    ch.xyz(); //valid
  }
}
```

Method overriding

Declaring a method in **sub class** which is already present in **parent class** is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. In this case the method in parent class is called overridden method and the method in child class is called overriding method. In this guide, we will see what is method overriding in Java and why we use it.

Method Overriding Example

Lets take a simple example to understand this. We have two classes: A child class Boy and a parent class Human. The `Boy` class extends `Human` class. Both the classes have a common method `void eat()`. Boy class is giving its own implementation to the `eat()` method or in other words it is overriding the `eat()` method.

The purpose of Method Overriding is clear here. Child class wants to give its own implementation so that when it calls this method, it prints Boy is eating instead of Human is eating.

```
class Human{
   //Overridden method
   public void eat()
   {
      System.out.println("Human is eating");
   }
}
class Boy extends Human{
   //Overriding method
   public void eat(){
      System.out.println("Boy is eating");
   }
   public static void main( String args[]) {
      Boy obj = new Boy();
      //This will call the child class version of eat()
      obj.eat();
   }
}
```

Output:

```
Boy is eating
```

Advantage of method overriding

The main advantage of method overriding is that the class can give its own specific implementation to a inherited method **without even modifying the parent class code**.

This is helpful when a class has several child classes, so if a child class needs to use the parent class method, it can use it and the other classes that want to have different implementation can use overriding feature to make changes without touching the parent class code.

Method Overriding and Dynamic Method Dispatch

Method Overriding is an example of runtime polymorphism. When a parent class reference points to the child class object then the call to the overridden method is determined at runtime, because during method call which method(parent class or child class) is to be executed is determined by the type of object. This process in which call to the overridden method is resolved at runtime is known as dynamic method dispatch. Lets see an example to understand this:

```
class ABC{
   //Overridden method
   public void disp()
   {
        System.out.println("disp() method of parent class");
   }
}
class Demo extends ABC{
   //Overriding method
   public void disp(){
        System.out.println("disp() method of Child class");
   }
   public void newMethod(){
        System.out.println("new method of child class");
   }
   public static void main( String args[]) {
        /* When Parent class reference refers to the parent class object
         * then in this case overridden method (the method of parent class)
         *  is called.
         */
        ABC obj = new ABC();
        obj.disp();

        /* When parent class reference refers to the child class object
         * then the overriding method (method of child class) is called.
         * This is called dynamic method dispatch and runtime polymorphism
         */
        ABC obj2 = new Demo();
        obj2.disp();
   }
}
```

Output:

```
disp() method of parent class
disp() method of Child class
```

In the above example the call to the disp() method using second object (obj2) is runtime polymorphism (or dynamic method dispatch).
Note: In dynamic method dispatch the object can call the overriding methods of child class and all the non-overridden methods of base class but it cannot call the methods which are newly declared in the child class. In the above example the object `obj2` is calling the `disp()`. However if you try to call the `newMethod()` method (which has been newly declared in Demo class) using obj2 then you would give compilation error with the following message:

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problem: The method xyz() is undefined for the type ABC
```

Rules of method overriding in Java

1. Argument list: The argument list of overriding method (method of child class) must match the Overridden method(the method of parent class). The data types of the arguments and their sequence should exactly match.
2. Access Modifier of the overriding method (method of subclass) cannot be more restrictive than the overridden method of parent class. For e.g. if the Access Modifier of parent class method is public then the overriding method (child class method) cannot have private, protected and default Access modifier,because all of these three access modifiers are more restrictive than public.
   For e.g. This is **not allowed** as child class disp method is more restrictive(protected) than base class(public)

```
class MyBaseClass{
   public void disp()
   {
       System.out.println("Parent class method");
   }
}
class MyChildClass extends MyBaseClass{
   protected void disp(){
      System.out.println("Child class method");
   }
   public static void main( String args[]) {
      MyChildClass obj = new MyChildClass();
      obj.disp();
   }
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problem: Cannot reduce the visibility of the inherited method from
MyBaseClass
```

However this is perfectly valid scenario as public is less restrictive than protected. Same access modifier is also a valid one.

```
class MyBaseClass{
   protected void disp()
   {
       System.out.println("Parent class method");
   }
}
class MyChildClass extends MyBaseClass{
   public void disp(){
      System.out.println("Child class method");
   }
   public static void main( String args[]) {
      MyChildClass obj = new MyChildClass();
      obj.disp();
   }
}
```

Output:

```
Child class method
```

17. private, static and final methods cannot be overridden as they are local to the class. However static methods can be re-declared in the sub class, in this case the sub-class method would act differently and will have nothing to do with the same static method of parent class.
18. Overriding method (method of child class) can throw unchecked exceptions, regardless of whether the overridden method(method of parent class) throws any exception or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. We will discuss this in detail with example in the upcoming tutorial.
19. Binding of overridden methods happen at runtime which is known as dynamic binding.
20. If a class is extending an abstract class or implementing an interface then it has to override all the abstract methods unless the class itself is a abstract class.

Super keyword in Method Overriding

The super keyword is used for calling the parent class method/constructor. `super.myMethod()` calls the myMethod() method of base class while `super()` calls the constructor of base class. Let’s see the use of super in method Overriding.

As we know that we we override a method in child class, then call to the method using child class object calls the overridden method. By using super we can call the overridden method as shown in the example below:

```
class ABC{
   public void myMethod()
   {
        System.out.println("Overridden method");
   }
}
class Demo extends ABC{
   public void myMethod(){
        //This will call the myMethod() of parent class
        super.myMethod();
        System.out.println("Overriding method");
   }
   public static void main( String args[]) {
        Demo obj = new Demo();
        obj.myMethod();
   }
}
```

Output:

```
Class ABC: mymethod()
Class Test: mymethod()
```

As you see using super keyword, we can access the overriden method.