JAVA BASICS

Java Packages

There are lot of class with different frameworks (servlet, JDBC) and external libraries as well. We separate their Class by their working, usability then put into particular folders.

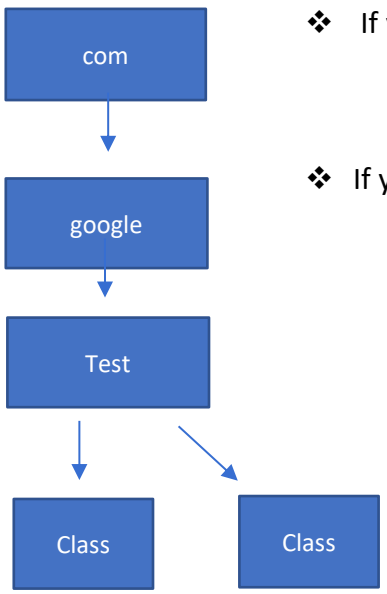
Eg : All IO Classes put into IO package.
All networking classes put into .net package.
All database classes put into sql package.

Like we grouping our songs into different playlists.

If you creating a package it's name should be unique.

Package name must be the mirror of domain name.

Google.com → com.google



- If you want to get all classes of test….
  Com.google.test. *

- If you want to get all classes of google….
  Com.google. *

Access Modifiers

In the Class you can't use private, protected and you can use only final, abstract, public, default.

If you have inner class you can use private.

If you want to access your class outside of the package make sure it is public.

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Static keyword

If you want your method without using any object we use static keyword.
Can't use non static variable in static method.
Static variable can use in non-static method.

Ex 01:

```
class variable {
   String studentName;
   static String TeacherName;

   public void var(){
      System.out.println("Student name is " + this.studentName);
      System.out.println("Teacher name is " + TeacherName);
   }
}


public class StaticVariable {

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

```
        variable.TeacherName = "scene";

        variable var1 = new variable();

        var1.studentName = "Janith";
        var1.var();

        System.out.println();

        variable var2 = new variable();

        var2.studentName = "John";
        var2.var();
    }
}
```

output :

Student name is Janith
Teacher name is scene

Student name is John
Teacher name is scene

++

Ex 02 :

```
class static_2{

    static String nameOfTeacher01;
    static int ageOfStudent;

    static{
        nameOfTeacher01 = "janith";
        ageOfStudent = 90;
        System.out.println("yyy");
    }

    public static_2(){
        System.out.println("xxx");
    }


    public void grading(String sub){
        System.out.println("The " + nameOfTeacher01 + " is the lecture , who teach the " + sub + " for the " +
ageOfStudent);
    }
}
```

```java
public class static_q1 {

   public static void main(String args[]){

      static_2 classing = new static_2();

      static_2.nameOfTeacher01 = "Dani daniels";
      static_2.ageOfStudent = 17;

      classing.grading("Mathematics");

      classing.grading("Science");

      static_2.ageOfStudent = 19;

      classing.grading("Bio");
   }

}
```

output :

yyy
xxx
The Dani daniels is the lecture , who teach the Mathematics for the 17
The Dani daniels is the lecture , who teach the Science for the 17
The Dani daniels is the lecture , who teach the Bio for the 19

++

Ex 03:

```java
class Something_Static{
   String name;
   static int age = 23;

   public Something_Static(){
      name = "janith";
      System.out.println("Constructor : " + name + " : " + age);
   }

 public void something(){
      name = "John";
      age = 19;
      System.out.println("method : " + name + " : " + age);
   }

   static{
      System.out.println("Static : " + age);
```

```
    }

    public void something2(){
        name = "Scene";
        System.out.println("method2 : " + name + " : " + age);
    }
}
public class Static_More {
    public static void main(String[] args) {
        Something_Static obj = new Something_Static();
        Something_Static.age =112;
        obj.something();
        obj.something2();
    }
}
```

output :

```
Static : 23
Constructor : janith : 23
method : John : 19
method2 : Scene : 19
```

++

The static class loading only once because the static method loading only once.

When you creating two objects from one class and run constructure, then the constructure calls times but your class is loading only once. That is why static block loading only once. Static block executes before the main method loads.
This will get call as soon as class is loading to JVM.

Static : Need a class to call variables and methods.
Non-static : Need object to call variables and methods.

Relationship between object and constructor

Once you created object of a class the constructor will run at once. You can count the objects you created using it's constructor…

Ex 04 -> **Counting objects**:

```
class counting{
   static int i = 0;
   public counting(){
      i++;
   }
   public void counter(){
      System.out.println(i);
   }
}
public class count_objects {
   public static void main(String[] args) {
      counting obj1 = new counting();
      counting obj2 = new counting();
      obj1.counter();
   }
}
```

output:

2

++

Ex 05 -> **Counting Methods**:

```
class countMethods{
   static int i = 2;

   public countMethods(){
      i = i + 1 ;
   }
   public void method1() {
      System.out.println(i);
   }
}


public class counting_Methods {
```

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

      countMethods obj1 = new countMethods();
      countMethods obj2 = new countMethods();
      obj1.method1();
   }

}
```

output:

4

++

Encapsulation

How to implement encapsulation in java:

1) Make the instance variables private so that they cannot be accessed directly from outside the class. You can only set and get values of these variables through the methods of the class.
2) Have getter and setter methods in the class to set and get the values of the fields.

```
class EncapsulationDemo{
   private int ssn;
   private String empName;
   private int empAge;

   //Getter and Setter methods
   public int getEmpSSN(){
      return ssn;
   }

   public String getEmpName(){
      return empName;
   }

   public int getEmpAge(){
      return empAge;
   }

   public void setEmpAge(int newValue){
      empAge = newValue;
   }
```

```
   public void setEmpName(String newValue){
      empName = newValue;
   }

   public void setEmpSSN(int newValue){
      ssn = newValue;
   }
}
public class EncapsTest{
   public static void main(String args[]){
      EncapsulationDemo obj = new EncapsulationDemo();
      obj.setEmpName("Mario");
      obj.setEmpAge(32);
      obj.setEmpSSN(112233);
      System.out.println("Employee Name: " + obj.getEmpName());
      System.out.println("Employee SSN: " + obj.getEmpSSN());
      System.out.println("Employee Age: " + obj.getEmpAge());
   }
}
```

output:

```
Employee Name: Mario
Employee SSN: 112233
Employee Age: 32
```

++

Advantages of encapsulation

1. It improves maintainability and flexibility and re-usability: for e.g. In the above code the implementation code of `void setEmpName(String name)` and `String getEmpName()` can be changed at any point of time. Since the implementation is purely hidden for outside classes they would still be accessing the private field empName using the same methods (`setEmpName(String name)` and `getEmpName()`). Hence the code can be maintained at any point of time without breaking the classes that uses the code. This improves the re-usability of the underlying class.
2. The fields can be made read-only (If we don't define setter methods in the class) or write-only (If we don't define the getter methods in the class). For e.g. If we have a field(or variable) that we don't want to be changed so we simply define the variable as private and instead of set and get both we just need to define the get method for that variable. Since the set method is not present there is no way an outside class can modify the value of that field.

3. User would not be knowing what is going on behind the scene. They would only be knowing that to update a field call set method and to read a field call get method but what these set and get methods are doing is purely hidden from them.

Inheritance

After you done a project you need to add some features on it but without changing the file… It is not possible but you can create new file with new features you want and add both files together using Inheritance.

Ex: If have Nokia 2.3 and the Nokia Company needs to add new features on it. They extend the new features to the older one and release it as a new phone.

There are several types of Inheritance:

01. Single -> A extends B
02. Multilevel -> A extends B and B extends C
03. Multiple -> A extends B, C (not possible in java)

In Inheritance when you create a object of subclass, automatically it calls constructor of SUPER Class.

Ex 01:

```
class A{
    public A(){
        System.out.println("Im A");
    }

    public A(int i){
        System.out.println("I'm father of A");
    }

    public void x(){
        System.out.println("This is new method 1");
    }
}


class B extends A{
    public B(){
        System.out.println("I'm B");
    }
```

```
    public B(int j){
        System.out.println("I'm father of B");
    }

}

public class inheritanceTwo {

    public static void main(String args[]){

        B obj1 = new B();
        obj1.x();
    }
}
```

output:

```
Im A
I'm B
This is new method 1
```

++

Remember that once you created object of the inherited class(children class) in main method , it is print output of the default constructor of parent class with it's own output.

```
abstract class Behaviours {
    abstract void sound();
    abstract void color();
}

class Dog extends Behaviours {
    @Override
    public void sound(){
        System.out.println("baaw baaaw");
    }
    @Override
    public void color(){
        System.out.println("black and white");
    }
}

class Cat extends Behaviours {
    @Override
    public void sound(){
```

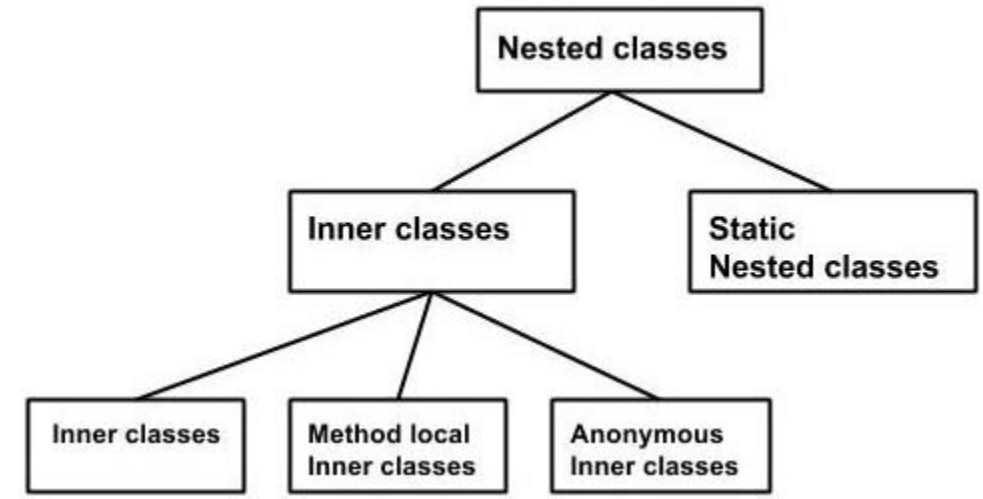
```
        System.out.println("maaeeeaaw");
    }
    @Override
    public void color(){
        System.out.println("pol sambola pata");
    }
}

public class soundsOfAnimals {

    public static void main(String[] args) {
        Behaviours doggy1 = new Dog();
        doggy1.sound();

        Behaviours doggy2 = new Cat();
        doggy2.color();
    }
}
```

Output:

baaw baaaw
pol sambola pata

++

Inner Classes



Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier **private**, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class.

Inner classes are of three types depending on how and where you define them. They are −

- Inner Class
- Method-local Inner Class
- Anonymous Inner Class

Inner Class

Creating an inner class is quite simple. You just need to write a class within a class. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

Following is the program to create an inner class and access it. In the given example, we make the inner class private and access the class through a method.

when Inner Class is non-static and its methods non-static too….

Ex 01:

```
class Outer_Demo {

   int num;

   // inner class
   private class Inner_Demo {
      public void print() {
         System.out.println("This is an inner class");
      }
   }

   // Accessing he inner class from the method within
   void display_Inner() {
      Inner_Demo inner = new Inner_Demo();
      inner.print();
   }

}
```

```
public class My_class {

   public static void main(String args[]) {
      // Instantiating the outer class
      Outer_Demo outer = new Outer_Demo();

      // Accessing the display_Inner() method.
      outer.display_Inner();
   }
}
```

output:

This is an inner class

++

Ex 02:

```
class Outer{
   public void show(){
      System.out.println("This is the outer class");
   }

   class innerOfOuter{
      public void something(){
         System.out.println("This is Inner class of the outer class");
      }
   }
}
public class OuterNonStatic{
   public static void main(String args[]){

      Outer obj1 = new Outer();
      obj1.show();

      Outer.innerOfOuter obj2 = obj1.new innerOfOuter();
      obj2.something();
   }
}
```

output:

This is the outer class
This is Inner class of the outer class

++

Method-local Inner Class

In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.

A method-local inner class can be instantiated only within the method where the inner class is defined. The following program shows how to use a method-local inner class.

Ex 03:

```
public class Outerclass {
   // instance method of the outer class
   void my_Method() {
      int num = 23;

      // method-local inner class
      class MethodInner_Demo {
         public void print() {
            System.out.println("This is method inner class "+num);
         }
      } // end of inner class

      // Accessing the inner class
      MethodInner_Demo inner = new MethodInner_Demo();
      inner.print();
   }

   public static void main(String args[]) {
      Outerclass outer = new Outerclass();
      outer.my_Method();
   }
}
```

output:

This is method inner class 23

++

Anonymous Inner Class

An inner class declared without a class name is known as an **anonymous inner class**. In case of anonymous inner classes, we declare and instantiate them at the same time. Generally, they are used whenever you need to override the method of a class or an interface. The syntax of an anonymous inner class is as follows

The following program shows how to override the method of a class using anonymous inner class.

Ex 04:

```
abstract class AnonymousInner {
   public abstract void mymethod();
}

public class Outer_class {

   public static void main(String args[]) {
      AnonymousInner inner = new AnonymousInner() {
         public void mymethod() {
            System.out.println("This is an example of anonymous inner class");
         }
      };
      inner.mymethod();
   }
}
```

output:

This is an example of anonymous inner class

++

Anonymous Inner Class as Argument

Generally, if a method accepts an object of an interface, an abstract class, or a concrete class, then we can implement the interface, extend the abstract class, and pass the object to the method. If it is a class, then we can directly pass it to the method.

But in all the three cases, you can pass an anonymous inner class to the method. Here is the syntax of passing an anonymous inner class as a method argument

Ex 04

```
interface Message {
   String greet();
}

public class Passing_Args_Class {

   // method which accepts the object of interface Message
   public void displayMessage(Message m){
      System.out.println(m.greet() + ", This is an example of anonymous inner class as an argument");
   }

   public static void main(String args[]) {
      // Instantiating the class
      Passing_Args_Class obj = new Passing_Args_Class();
```

```
        // Passing an anonymous inner class as an argument
        obj.displayMessage(new Message() {
            public String greet() {
                return "Hello";
            }
        });
    }

}
```

output:

Hello, This is an example of anonymous inner class as an argument

++

Super Method

If you create object of Subclass its automatically call the constructor of super class.

Ex 01:

```
class Axa{
    public Axa(){
        System.out.println("this is constructor of A");
    }
    public Axa(int k){
        System.out.println("this is constructor A with para");
    }
    public void alter(){
        System.out.println("Alter method in A ");
    }
}

class Bxa extends Axa{

    public Bxa(){
        super();
        System.out.println("this is constructor of B");
    }

    public Bxa(int h){
        super(1);
        System.out.println("this is constructor B with para");
    }

    @Override
    public void alter(){
        super.alter();
        System.out.println("Alter method in B ");
    }
```

```
}
public class methoding {
   public static void main(String[] args) {
      Axa object = new Bxa(1);
      object.alter();
   }
}
```

output:

```
this is constructor A with para
this is constructor B with para
Alter method in A
Alter method in B
```

++

Super Method

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

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor

Ex 01:

```
class riding2{
   public void method1(){
      System.out.println("Processing method 1");
   }
}

class riding3 extends riding2{
   @Override
   public void method1(){
      System.out.println("processing method 2");
```

```
    }
}
public class riding1 {
    public static void main(String args[]){
        riding3 obj1 = new riding3();
        obj1.method1();
    }
}
```

In order to access the super class's overridden method ->

```
@Override
public void method1(){
    super.method1();
    System.out.println("processing method 2");
}
```

++

Dynamic Method dispatch

Compiling a **Java** program means taking the programmer-readable text in your program file (also called source code) and converting it to bytecodes, which are platform-independent instructions for the **Java** VM.

Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
- At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed
- A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time

Ex 01:

```
class A{
   public void show(){
      System.out.println("This is A");
   }
}

class B extends A{
   @Override
   public void show(){
      System.out.println("This is B");
   }
   public void display(){
      System.out.println("Displaying....");
   }
}

class C extends A{
   @Override
   public void show(){
      System.out.println("This is C");
   }
}
public class dispatching {
   public static void main(String args[]){
      // there are compile time polymorphism and runtime polymorphism .. and runtime polymorphism  can
decide whid method
      // should call
      A obj1 = new B();
      obj1.show();

      obj1 = new C();
      obj1.show();   //dynamic method dispatching

   }
}
```

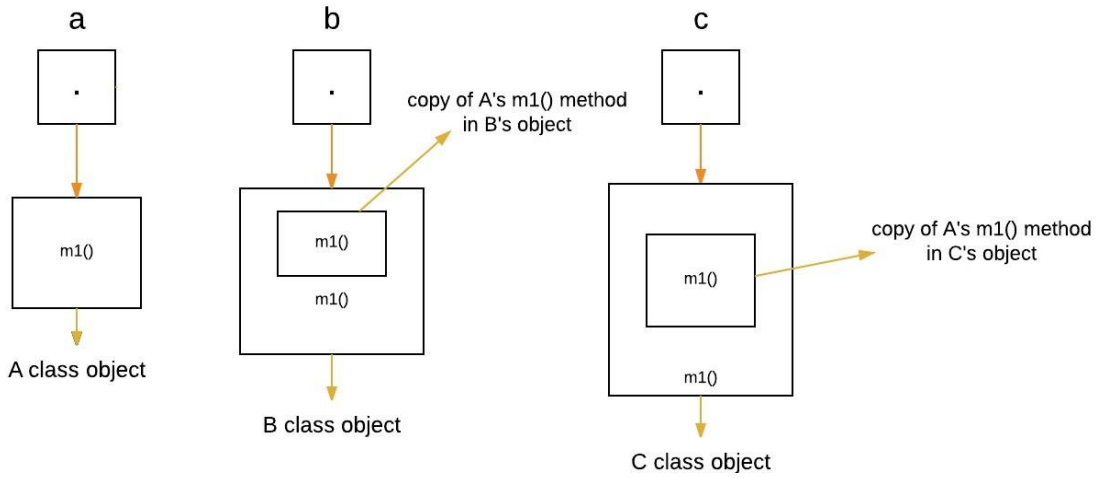
output:

```
This is B
This is C
```

++

1.Inside the main () method in Dispatch class, initially objects of type A, B, and C are declared.

```
A obj1 = new A (); // object of type A

B obj2 = new B (); // object of type B

C obj3 = new C (); // object of type C
```



2. Now a reference of type A, called ref, is also declared, initially it will point to null

4. Now we are assigning a reference to each **type of object** (either A's or B's or C's) to *ref*, one-by-one, and uses that reference to invoke show (). As the output demonstrating, the version of show () executed is determined **by the type of object being referred to at the time of the call.**

```
ref = a; // r refers to an A object

ref. show (); // calling A's version of m1()
```

Wrapper Class

Java is pure Object Oriented, but there are primitive data types came from C language. Java OOP says everything should be objects but when we create primitive data type like int it is not an object type. That is why java is 99.9% OO language.

Integer I = new Integer (5); Object

- When you put primitive variable into Object (Wrapper Class) that calls Boxing in OOP.

```
Int I = 5;
Integer j = new Integer (i);
```

- Taking out a value and assign it to primitive type variable calls Unboxing.

```
Int i = 5;
Integer j = new Integer (i);
Int k = j . intValue ();
```

- Also, the java compiler can do that Object creation automatically when boxing. That calls Autoboxing.

```
Int i = 6;
Integer j = i;
```

- When Autounboxing you don't have to specify the "intvalue" part. Compiler automatically do it for you.

```
Integer j = new Integer (i);
Int k = j;
```

Primitive data types are faster than wrapper classes. But there are certain frameworks, they are only work with wrapper classes.

When you need assign String value as int value there is method call parseInt ().

```
String str = "123";
Int n = Integer.parseInt(str);
```

Abstract Keyword

When you need to terminate the creating Objects because improve security, we need Abstract keyword. Once you create your class abstract no one can access it with creating Objects.

If you need declare (mot define) a method, name it as abstract…

When you having abstract method also the class will be abstract method can have non-abstract methods. There is a path you can create objects of abstract class with using subclass of it.

All abstract methods in abstract class must be override in subclass. Also, you can define the methods and not compulsory to override it on subclass.

Ex 01:

```
abstract class ABCD{
   abstract void method_ABCD();

   abstract void LMNO();

   public void non_abstract_method(){

   }
}

// If you declaring a non-abstract method in abstract class not compulsory to override it on by-extended
class..

interface PQRS {
   int i=0;

   default void method_PQRS(){

   }
   void second_Method();
}

class unknown extends ABCD{
   @Override
   void method_ABCD() {

   }

   @Override
   void LMNO() {

   }
}
```

Ex 02:

```
abstract class abstract1{
   public void xyz(){
      System.out.println("This is xyz method");
   }
   public abstract void omn();
}

class abstract2 extends abstract1{
   public void abc(){
      System.out.println("This is abc method");
   }
   @Override
   public void omn(){
      System.out.println("Abstract method by abstract1");
```

```
  }
}

class abstract3{
  public void lmn(){
    System.out.println("This is lmn method");
  }
}

public class abstract_Class {
  public static void main(String args[]){
    abstract2 bj1 = new abstract2();
    bj1.xyz();
  }
}
```

output:

This is xyz method

++

Polymorphism

Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So, polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Polymorphism = method Overriding + Overloading;

Runtime Polymorphism in Java

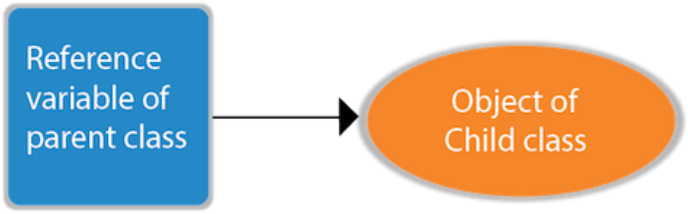
Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:




```
class A {}
class B extends A {}
A a=new B () ;//upcasting
```

For upcasting, we can use the reference variable of class type or an interface type. For Example:

```
interface I {}
class A {}
class B extends A implements I {}
```

Here, the relationship of B class would be:

```
B IS-A A
B IS-A I
B IS-A Object
```

Since Object is the root class of all classes in Java, so we can write B IS-A Object

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

Ex 01:

```
class Bike{
   void run(){System.out.println("running");}
}
class Splendor extends Bike{
   void run(){System.out.println("running safely with 60km");}

   public static void main(String args[]){
      Bike b = new Splendor();//upcasting
      b.run();
   }
}
```

Final Keyword

Final keyword with variable:

IF you make variable as final you are not allowed to change the value after you assign at once. It becomes constant and use capital letters in Constant.

Final keyword with Class:

If you make your class as final, it can't be a super class anymore because it can't extend.

Final keyword with method:

You can't override that method from super class.

Anonymous Object

Yes, we can use a method on an object without assigning it to any reference.

A obj = new A ();

Obj.show (); =====➔ new A (). show ();

```
public class Tester {
   public String message(){
      return "Hello World!";
   }
   public static void main(String[] args) {
      System.out.println(new Tester().message());
   }
}
```

When you do like this, will not create any data in stack memory and taking place in heap memory. You want to use object only once, just user anonymous object…