OOPS Concepts in JAVA

Inheritance: It is a mechanism where child class acquire all the properties & behaviour of parent class.

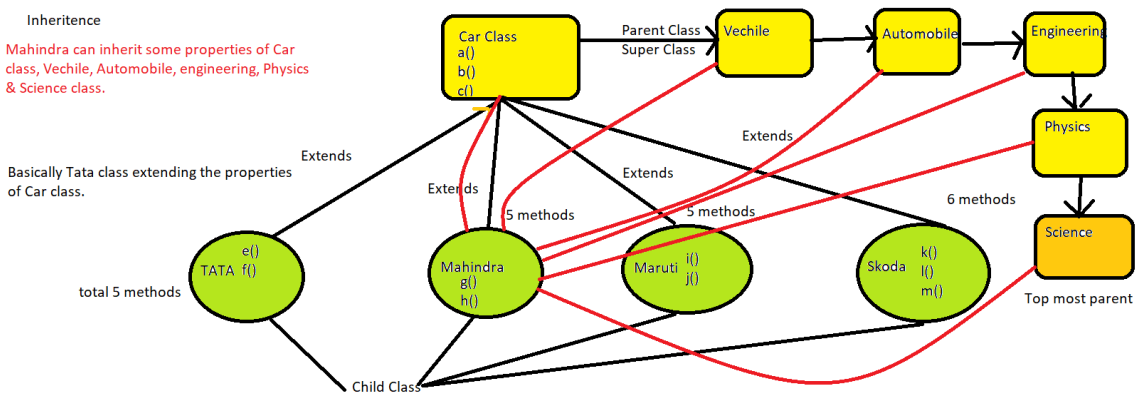
The idea behind the inheritance concept is that you can create new child classes, that are built upon existing parent classes. So, when you inherit the parent class then you can re-use the methods & variable of parent class. And you can add new methods & variables in your child class as well. We use extends keywords to acquire all the properties & behaviour of parent class.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why we use Inheritance?

For Method Overriding (To achieve run time polymorphism)

For Code reusability.



Method Overriding: When same methods are available in parent class as well as in child class with same name & same no of arguments. It is called method overriding. And here start method is overridden method.

Note: Child class object can be referred by the parent class**(CAR)** reference variable.--- Dynamic polymorphism -- Run time polymorphism. We can call only Parent class methods & Common methods of parents & child class. We cannot call the methods are present in child class only by using the reference variable of parent class. Mahindra class **(Child class)** will not allow to access. Only Mahindra class reference variable can access the methods are available in Mahindra class.

Parent Class

Car.java | Mahindra.java | TestCar.java | Vechile.java

```java
package javaConcepts.Inheritance;

public class Vechile {

    public void engine() {

        System.out.println("Vechile-------engine");
    }

}

```

Car.java | Mahindra.java | TestCar.java | Vechile.java

```java
package javaConcepts.Inheritance;

public class Car extends Vechile {

    public void start() {

        System.out.println("Car---- Starting");

    }
public void run() {

        System.out.println("Car---- Running");

    }

public void safety() {

        System.out.println("Car---- Safety");

    }

}
```

Car.java | *Mahindra.java | TestCar.java | Vechile.java

```java
package javaConcepts.Inheritance;

public class Mahindra extends Car {

public void start() {

        System.out.println("Mahindra---- Starting");

    }
public void run() {
    System.out.println("Mahindra---- Running");
}
public void safety() {
        System.out.println("Car---- Safety");
    }
public void xuv300() {
    System.out.println("Mahindra XUV300---- Starting");
}
public void xuv700() {
    System.out.println("Mahindra xuv700---- Safety");
}

}
```

Car.java | *Mahindra.java | *TestCar.java | Vechile.java

```java
package javaConcepts.Inheritance;
public class TestCar extends Mahindra {
    public static void main(String[] args) {
        Mahindra m = new Mahindra();
        // When same methods are available in parent class as well as in child class with same name & same no of arguments
        // It is called method overriding. And here start method is overridden method.
        m.start();
        m.xuv300();
        m.safety();
        m.engine();
        System.out.println("----------------");
        Car c = new Car();
        c.start();
        c.safety();
        System.out.println("*****************");
        Car c1 = new Mahindra(); // Child class object can be referred by the parent class reference variable.--- Dynamic polymorphism --
        c1.start(); // We can call only reference class methods & Common methods of parents & child class.
        c1.safety();// We cannot call the methods are present in child class only. Mahindra class will not allow to access. Only Mahindra
                    // variable can access the methods are available in Mahindra class.
        c1.run();
```

Console | Debug Shell | TestNG

```
<terminated> TestCar [Java Application] C:\Program Files\Java\jre1.8.0_291\bin\javaw.exe (06-Aug-2022, 6:00:13 am)
Mahindra---- Starting
Mahindra XUV300---- Starting
Car---- Safety
Vechile-------engine
----------------
Car---- Starting
Car---- Safety
*****************
Mahindra---- Starting
Car---- Safety
Mahindra---- Running
```

Up Casting & Down Casting:

```java
        //Up Casting ( Child class object referred by parent class reference variable)
        Car c1 = new Mahindra(); // Child class object can be referred by the parent class reference variable.--- Dynamic polymorphism --
        c1.start(); // We can call only reference class methods & Common methods of parents & child class.
        c1.safety();// We cannot call the methods are present in child class only. Mahindra class will not allow to access. Only Mahindra
        c1.run();   // variable can access the methods are available in Mahindra class.
        //Down casting
        Mahindra m1 =(Mahindra) new  Car(); // Big thing (Car class object) cannot be fit into small thing (Mahindra class reference varia
        // It will give the ClassCastException
```

Console | Debug Shell | TestNG

```
<terminated> TestCar [Java Application] C:\Program Files\Java\jre1.8.0_291\bin\javaw.exe (06-Aug-2022, 6:17:50 am)
Vechile-------engine
----------------
Car---- Starting
Car---- Safety
*****************
Mahindra---- Starting
Car---- Safety
Mahindra---- Running
Exception in thread "main" java.lang.ClassCastException: javaConcepts.Inheritance.Car cannot be cast to javaConcepts.Inheritance.Mahindra
        at javaConcepts.Inheritance.TestCar.main(TestCar.java:22)
```

Method Overloading: When a class has same methods having same name within the same class but having different no of arguments and different types of arguments, that is called method overloading.

Purpose of method overloading: If you want to perform same operation for different no of arguments or different types of arguments so it's better to use the same name of method. Because it's better to understand the behaviour of method with its name. And if you will write different name of methods so it will be difficult to understand the behaviour of method with its name.

MethodOverloading.java

```java
package javaConcepts;

public class MethodOverloading {
    public static void main(String[] args) {
        MethodOverloading ref = new MethodOverloading();
        ref.display("Himanshu");
        ref.sum(10);
        ref.sum(10, 20);
        ref.display(20);
        // Mehtod Overloading---> When the method name is same with different no of arguments and different type of arguments
        //                        within the same class.
    }
    public static void main(int[] args) {}
    public void display(int j) {    // 1 input param with same int type argument but different method name
        System.out.println("j is printing in display method: " + j);
    }
    public void display(String name) {  // 1 input param with different type of argument
        System.out.println("Name is printing: " + name);
    }
    public void sum(int i) {    // 1 input param
        System.out.println("SUm is printing in sum method: " + i);
    }
    public void sum(int i, int j) { // 2 input param
        System.out.println("Sum is printing: " + i +j);
    }

```

Console | Debug Shell | TestNG

```
<terminated> MethodOverloading [Java Application] C:\Program Files\Java\jre1.8.0_291\bin\javaw.exe (08-Aug-2022, 2:48:42 am)
Name is printing: Himanshu
SUm is printing in sum method: 10
Sum is printing: 1020
j is printing in display method: 20
```

 # Can we overload main method?

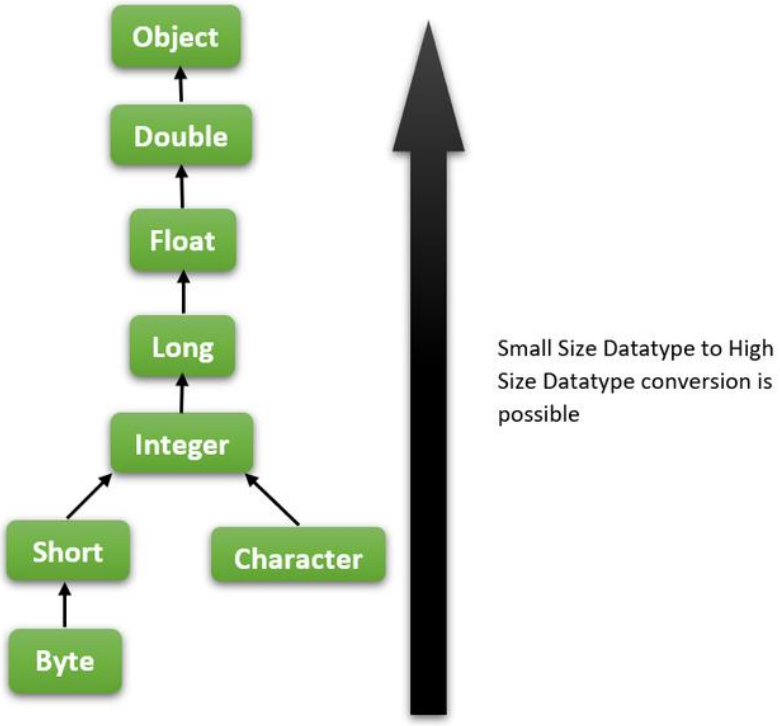
Yes, we can by changing its argument type. Here we are passing String[] & int [] as an argument.

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

```java
public static void main(int[] args)
```

Method Overloading with type promotion:

Note:- *This is important to remember is Automatic Type Promotion is only possible from small size datatype to higher size datatype but not from higher size to smaller size. i.e., integer to character is not possible.*



MethodOverloadingWithTypePromotion.java

```java
package javaConcepts;

public class MethodOverloadingWithTypePromotion {

    public static void main(String[] args) {

        MethodOverloadingWithTypePromotion obj = new MethodOverloadingWithTypePromotion();

        obj.print(20 );
        obj.print('a', 'b');
    }

    public void print(int i, int j) {
        System.out.println("char to int typpromotion Print method printing: " + i+j);

    }
    public void print(double k) {
        System.out.println("int to double print method printing: " + k);

    }

}
```

Console Debug Shell TestNG

```
<terminated> MethodOverloadingWithTypePromotion [Java Application] C:\Program Files\Java\jre1.8.0_291\bin\javaw.exe (08-Aug-20
int to double print method printing: 20.0
char to int typpromotion Print method printing: 9798
```

So, in above example we can see that how automatic type promotion is happening.

 # Data Hiding: Our internal data should not go out directly. Outside person should not access it directly. There should be some validation must be required before accessing the data. After validation only third person should able to access it.

Ex: If you want to access your Gmail inbox. So before accessing inbox, you must have to provide the User id & Pwd for validation. Once validation done then only you should able to access you inbox emails only.

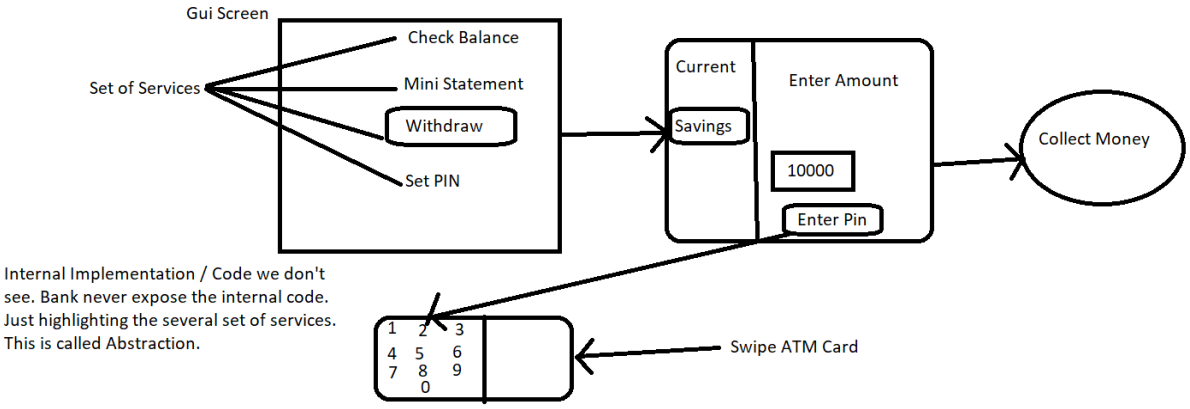
One more example for checking balance on Phone pe & google pay, you have to provide your UPI pin then only you can able to check your balance. Means Bank is hiding the balance from end user directly because of security reasons. Once you provide all your valid id or pwd then only you can see your balance. Sometimes OTP is required to validate the user authentication. Then only you can do login or make the transactions.

So by declaring the variable as private you can hide the data.

Private double balance;

 # Abstraction: Hiding internal implementation and just highlight set of services what we are offering is the concept of abstraction.

Example:



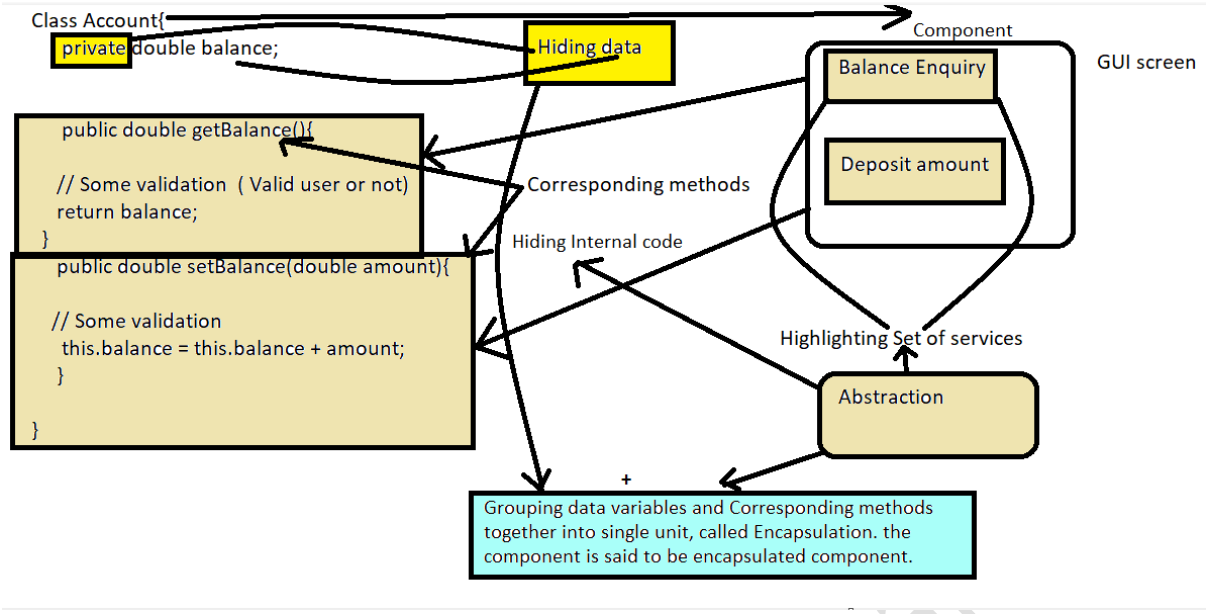
Note: The biggest advantage of abstraction is "Security" .

 # Encapsulation: The process of grouping data members and corresponding methods into a single unit is the concept of encapsulation.

Note: If any component follows data hiding & abstraction, that component is said to be encapsulated component.

Encapsulation = Data Hiding + Abstraction

Hiding data behind of methods is concept of encapsulation.



Method Hiding:

If it is overriding then method resolution is always taken care by JVM at runtime object. But in case of method hiding the method resolution is always take care by compiler based on reference type or variable.

If both methods are static then always method hiding will happen. If both are non-static methods then method overriding will happen.

So, method hiding is compile time polymorphism or static polymorphism or early binding. And method overriding is runtime polymorphism or dynamic polymorphism or late binding.

Example:

Parent.java | Child.java | TestMethodHiding.java

```java
package methodHiding;

public class Parent {

    public static void hide() {

        System.out.println("Parent printing");
    }

    public void override() {

        System.out.println("Parent Override method executing");
    }

}

```

Parent.java | Child.java | TestMethodHiding.java

```java
package methodHiding;

public class Child extends Parent {

    public static void  hide() {

        System.out.println("Child  printed");
    }

public void override() {

        System.out.println("child Override method executing");
    }

}

```

Parent.java | Child.java | TestMethodHiding.java

```java
package methodHiding;

public class TestMethodHiding extends Child {

    public static void main(String[] args) {
        Parent p = new Parent();
        p.hide();

        Parent obj = new Child();
        obj.hide();

        obj.override();


    }

}

```

Reference type or variable
hide () method is static so based on reference type it will print the hide() method of parent class. Means method execution is happening based of reference variable.

this is the non static method in parent & child class. it is overriding the parent class method. here method execution is happening based on object. child class object.

Final Output:

Parent.java Child.java TestMethodHiding.java

```java
package methodHiding;

public class TestMethodHiding extends Child {

    public static void main(String[] args) {
        Parent p = new Parent();
        p.hide();

        Parent obj = new Child();
        obj.hide();

        obj.override();


    }

}
```

Console Debug Shell TestNG

```
<terminated> TestMethodHiding [Java Application] C:\Program Files\Java\jre1.8.0
Parent printing
Parent printing
child Override method executing
```

If both hide() methods are non-static then see the output:

Parent.java Child.java TestMethodHiding.java

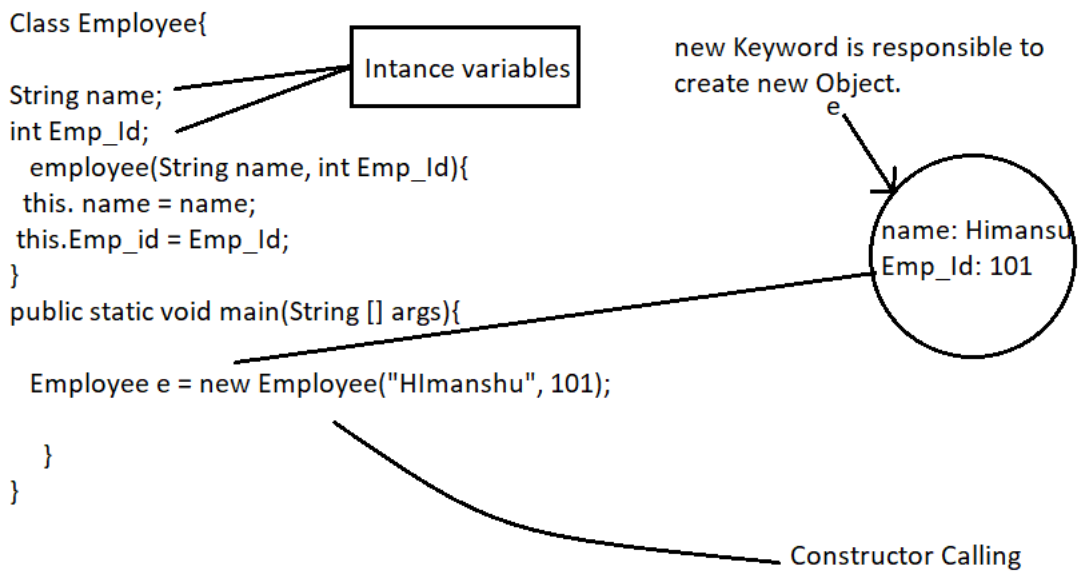
```java
package methodHiding;

public class TestMethodHiding extends Child {

    public static void main(String[] args) {
        Parent p = new Parent();
        p.hide();

        Parent obj = new Child();
        obj.hide();

        obj.override();


    }

}
```

Console Debug Shell TestNG

```
<terminated> TestMethodHiding [Java Application] C:\Program Files\Java\jre1.
Parent printing
Child  printed
child Override method executing
```

Now you can see the difference, It has override the parent class hide() method. now method execution is taken care by JVM based on runtime object. so it is executing the child class hide()

 # Constructor: Need of constructor: -→ Whenever we create the new object by using new Keyword then there must be some initialization must be required for our instance variables. So to initialize the object we need constructor.



To create Constructor:

1. **Constructor name must be same as class name.**
2. **A Constructor must have no explicit return type.**
3. **A java constructor cannot be abstract, final, static and synchronized.**

Note: We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

Every time an object is created using the new() keyword, at least one constructor is called. It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in java:

1. **Default Constructor (No arg Constructor): To provide the default values like Null for String, Zero for int variables.**
2. **Parameterized Constructor: It is used to provide the different values to different objects.**

Constructor with Super() & This()

Case 1: this() or Super() must be first statement in Constructor.

```
*Test.java
1  package constructor;
2
3  public class Test {
4
5      Test(){
6          System.out.println("Constructor Calling");
7          super();
8
9      }
10
11     public static void main(String[] args) {
12
13         Test t = new Test();
14     }
15
16 }
17
```

Constructor call must be the first statement in a constructor

Press 'F2' for focus

You can see above, this is not allowed in Java.

In screenshot below we can see the error is gone.

```
*Test.java
1  package constructor;
2
3  public class Test {
4
5      Test(){
6          super();
7          System.out.println("Constructor Calling");
8
9
10     }
11
12     public static void main(String[] args) {
13
14         Test t = new Test();
15     }
16
17 }
```

Case2: We cannot use super() & this() call in constructor together.

Test.java

```
1  package constructor;
2
3  public class Test {
4
5      Test(){
6          super();
7          this();
8          Syst
9
10
11     }
12
13     public static void main(String[] args) {
14
15         Test t = new Test();
16     }
17
18 }
```

Constructor call must be the first statement in a construc

Press 'F2' for fo

It is used to call super class or parent class conctructor.

It is calling current class constructor. That's why we cannot use both together.

Constructor Overloading:

Test.java | ConstructorOverloading.java

```
3  public class ConstructorOverloading {
4
5      ConstructorOverloading (double d){
6          this(20);
7      System.out.println("double arg constructor called");
8      }
9      ConstructorOverloading (int i){
10         this();
11         System.out.println("Integer arg Constructor called");
12     }
13     ConstructorOverloading (){
14         System.out.println("No arg Constructor called");
15     }
16
17
18
19     public static void main(String[] args) {
20
21         ConstructorOverloading c = new ConstructorOverloading(10.5);
22         ConstructorOverloading c1 = new ConstructorOverloading(25);
23         ConstructorOverloading c2 = new ConstructorOverloading();
24         }
```

3

2

1

No arg Constructor called

Integer arg construcotr called

double arg constructor called.

Integer arg.....

No arg.....

No arg.....

Console | Debug Shell | TestNG

```
<terminated> ConstructorOverloading [Java Application] C:\Program Files\Java\jre1.8.0_291\bin\javaw.
No arg Constructor called
Integer arg Constructor called
double arg constructor called
No arg Constructor called
Integer arg Constructor called
No arg Constructor called
```

 # Parent and child class constructor calls by Super(). What is happening internally, will see below:

*Parent.java Child.java

```
1 package ConsCallSuper;
2
3 public class Parent {
4         Parent(){
5         Super();
6 }       }
7
```

*Parent.java Child.java

```
1 package ConsCallSuper;
2
3 public class Child extends Parent {
4         Child(){}
5         Super();
6
7 }
8
```

This is all happening internally.
Matching contructor is available.
So there is no problem.

Case 2:

ParentOne.java ChildOne.java

```
1 package ConsCallSuper;
2
3 public class ParentOne {
4
5     ParentOne(){
6         Super();
7     }
8
9 }
10
```

Default Constructor

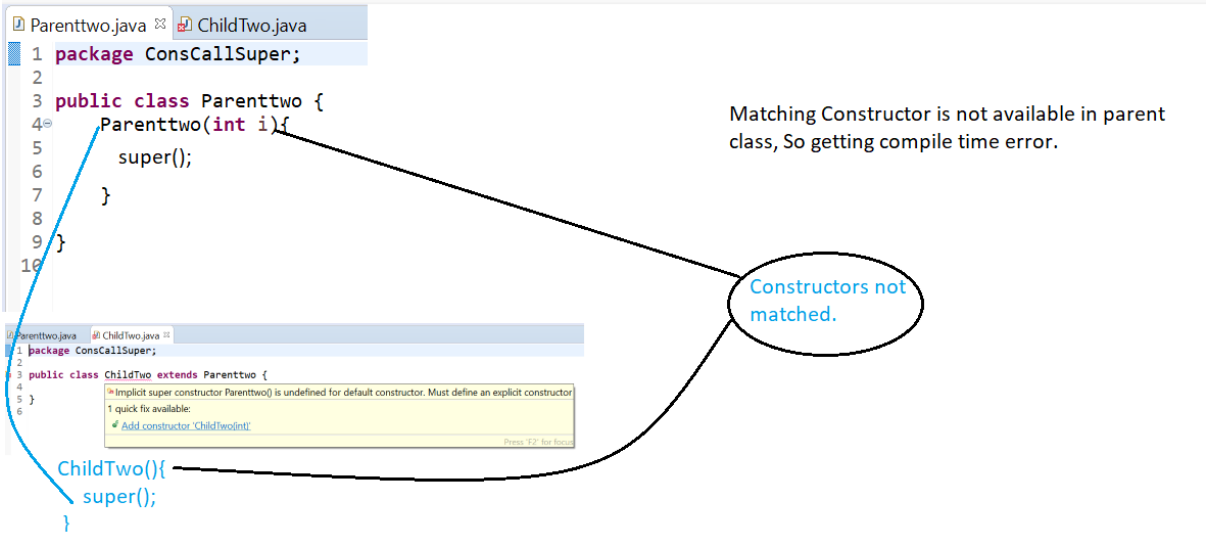
Matching Constructor is available. So happily code will compile. there is no problem at all.

This is all added by compiler internally.

ParentOne.java ChildOne.java

```
1 package ConsCallSuper;
2
3 public class ChildOne extends ParentOne{
4         ChildOne(){
5 }         SUper();
6         }
```

Case3:



 # Super: There are main 3 uses of super keyword in Java.

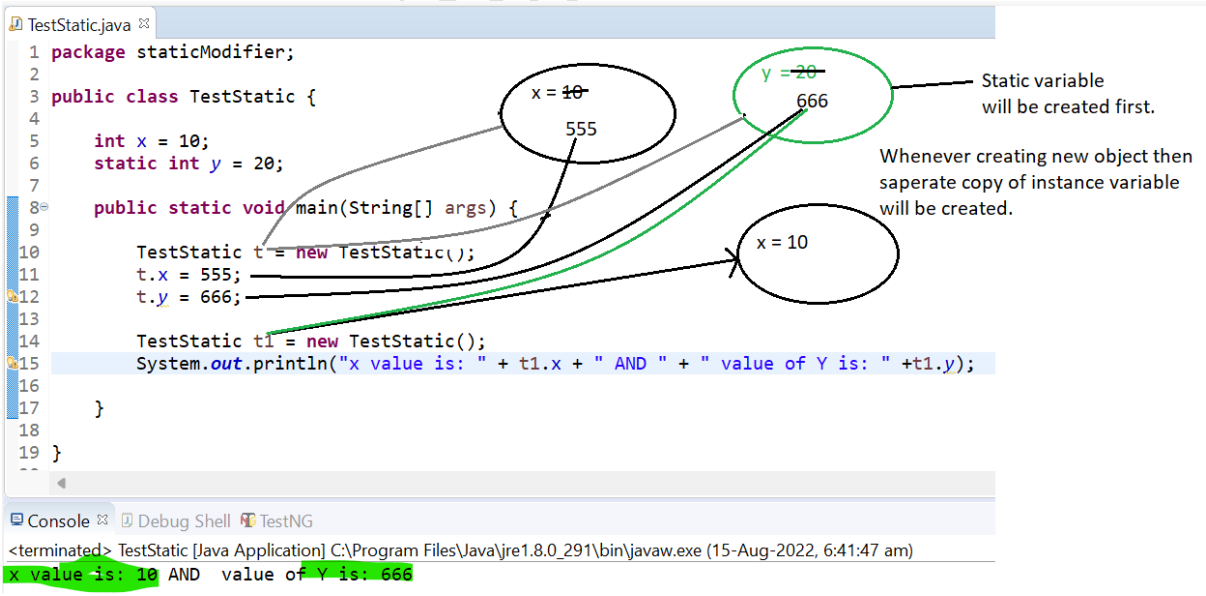
1. super can be used to refer immediate parent class instance variable.

2. super can be used to invoke immediate parent class method.

3. super() can be used to invoke immediate parent class constructor.

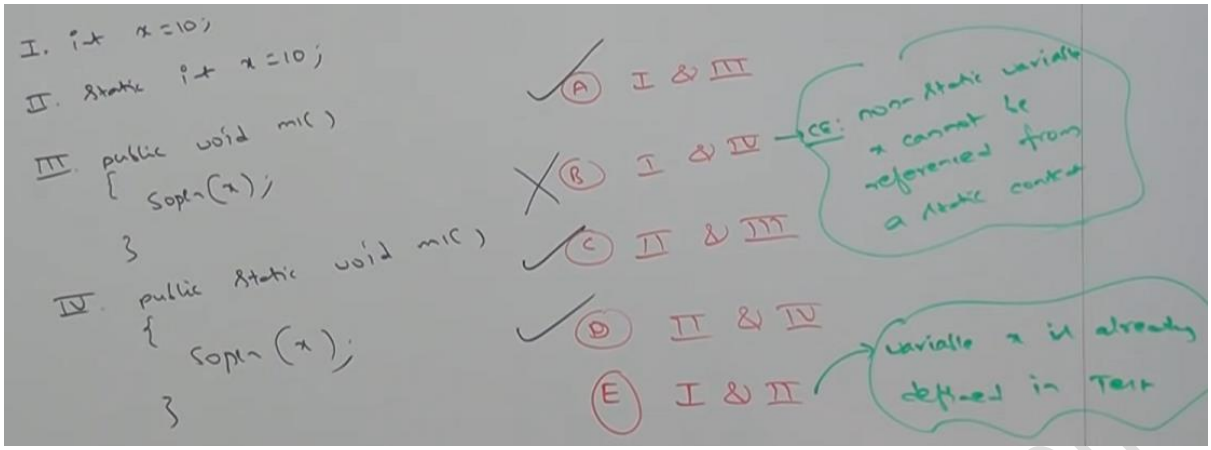
 # Static Keyword in Java: It is a modifier and applicable for methods & variables but not for classes. We can't declare top level class with static modifier but we can declare inner class as static. Such type of inner classes are called static nested classes.

Note: In the case of instance variables a separate copy will be created but in case of static variables a single copy will be created at class level and shared by every object of that class.



We can't access instance members directly from static area. But we can access from instance area directly. We can access static members from both static & instance area directly.

Q: Within the same class which of declaration we can take simultaneously ?



Duplicate method also name not allowed whether it is static or non static

TestStatic.java | CheckMethod.java

```
1  package staticModifier;
2
3  public class CheckMethod {
4
5      int x = 10;
6      static int y = 10;
7
8      public void show() {
9
10
11     }
12 public static vo
13
14
15     }
16
17     public static void main(String[] args) {
18
19
```

Duplicate method show() in type CheckMethod

1 quick fix available:

Rename method 'show' (Ctrl+2, R)

Press 'F2' for focus

Case 1: Over loading concept is also applicable for static methods including main method but JVM can always call String array argument main method only.

CheckMethod.java

```java
package staticModifier;

public class CheckMethod {

    public static void main(String[] args) {
        System.out.println("String array print");

    }
    public static void main(int[] args) {

        System.out.println("Int array print");

    }

}
```

Console | Debug Shell | TestNG

```
<terminated> CheckMethod [Java Application] C:\Program Files\Java\jre1.8.0
String array print
```

For Static methods overloading & inheritance concepts are applicable but overriding concept is not applicable but instead of overriding method hiding concept is applicable.

CheckMethod.java | Child.java

```java
package staticModifier;

public class CheckMethod {

    public static void main(String[] args) {
        System.out.println("Parent Main method");

    }


}
```

CheckMethod.java | Child.java

```java
package staticModifier;

public class Child extends CheckMethod {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Child main method");

    }

}
```

Console | Debug Shell | TestNG

```
<terminated> Child [Java Application] C:\Program Files\Java\jre1.8.0_291\bin\ja
Child main method
```

Static can be

1. Variable (also known as a class variable)

2. Method (also known as a class method)

3. Block

4. Nested class

 # Final: The final keyword in java is used to restrict the user. The java final keyword can be used in many contexts. Final can be:

1. **Variable (To stop value change):**

Without Final variable

Test.java

```java
package conceptFinal;

public class Test {
     int x = 100;
    void m() {

         x = 120;
        System.out.println(" Value changed " + x);
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.m();


    }

}

```

Console Debug Shell TestNG

```
<terminated> Test (1) [Java Application] C:\Program Files\Java\jre1.8.0_291\bi
 Value changed 120
```

With Final variable:

Test.java

```java
package conceptFinal;

public class Test {
     final int x = 100;
    void m() {

         x = 120;
        System.out.println(" Value changed " + x);
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.m();


    }

}
```

Console Debug Shell TestNG

```
<terminated> Test (1) [Java Application] C:\Program Files\Java\jre1.8.0_291\bin\javaw.exe (15-Aug-2022, 
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        The final field Test.x cannot be assigned

        at conceptFinal.Test.m(Test.java:7)
        at conceptFinal.Test.main(Test.java:13)
```

2. **Method (To stop method overriding)**

```java
package javaConcepts.Inheritance;

public class Car extends Vechile {

    public final void start() {

        System.out.println("Car---- Starting");

    }
public void run() {

        System.out.println("Car---- Running");

    }
```

```java
package javaConcepts.Inheritance;

public class Mahindra extends Car {

public void start() {

        Syst                                    ting");

    }
public void
    System.out.                                  );
}
```

Cannot override the final method from Car
1 quick fix available:
Remove 'final' modifier of 'Car.start(..)'
Press 'F2' for focus

3.
4. **Class (To stop Inheritance)**

```java
package javaConcepts.Inheritance;

public final class Car extends Vechile {

    public void start() {

        System.out.println("Car---- Starting");

    }
```

```java
package javaConcepts.Inheritance;

public class Mahindra extends Car {

public void start() {

        System.out.println("Ma

    }
public void run() {
```

The type Mahindra cannot subclass the final class Car
1 quick fix available:
Remove 'final' modifier of 'Car'
Press 'F2' for focus

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.

```java
package conceptFinal;

public class Blank {

    final int x;
    String name;

    Blank(){

        x = 10;
    }


    public static void main(String[] args) {

        Blank  b = new Blank();

            System.out.println(b.x);
    }

}
```

blank final variable

We can initialize it in Constructor only.

Console Debug Shell TestNG

```
<terminated> Blank [Java Application] C:\Program Files\Java\jre1.8.0_
10
```

Final Static variable with initialization

```java
package conceptFinal;

public class Blank {

    final static int x;
    String name;

    static{
        System.out.println("Static block called");
        x = 20;
    }


    public static void main(String[] args) {


            System.out.println(x);
    }

}
```

Call at the time of class loading
Static block is used to initialized
final static variables.

```
Console  Debug Shell  TestNG
<terminated> Blank [Java Application] C:\Program Files\Java\jre1.8.0_291\bir
Static block called
20
```

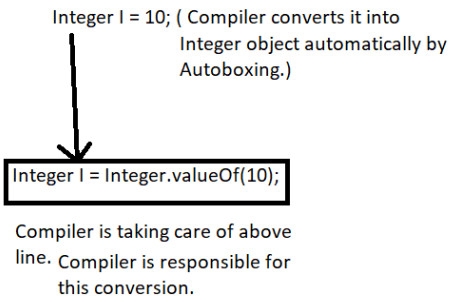
Wrapper Class: To wrap primitive data types into Java required object form some concept must be required that concept itself it is nothing but Wrapper classes.

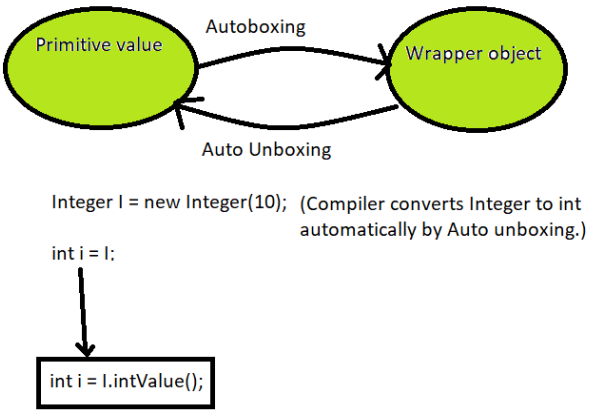
There is total 8 primitive data types in Java:

1. **int → Integer**
2. **boolean → Boolean**
3. **double → Double**
4. **float → Float**
5. **char → Character**
6. **byte → Byte**
7. **short → Short**
8. **long → Long**

Autoboxing: Automatic conversion from primitive to wrapper object by compiler is called autoboxing.

Auto Unboxing: Automatic conversion from wrapper object to primitive by compiler is called auto unboxing.





 # Abstract Class: Partially implemented classes are known as abstract class. We use abstract keyword to define the abstract class.

Note: Cannot create the object for abstract class. We'll get the compile time error if try to create the object of abstract class. "Cannot instantiate the type 'class name' ".

```
Blank.java   Test.java
1 package abstractConceptInterface;
2
3 public abstract class Test {
4
5     public static void main(String[] args) {
6         System.out.println("Abstract class");
7         Test t = new Test();
8
```

Cannot instantiate the type Test
Press 'F2' for focus

```
Console   Debug Shell   TestNG
<terminated> Test (2) [Java Application] C:\Program Files\Java\jre1.8.0_291\bin\javaw.exe (19-Aug-2022, 4:
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        Cannot instantiate the type Test

        at abstractConceptInterface.Test.main(Test.java:7)
```

Note: If a class contains at least one abstract method then it is compulsory to declare that class as abstract. Otherwise we will get the compile time error.

```
Blank.java   Test.java
1 package abstractConceptInterface;
2
3 public class Test {
4
5     public abstract void m();
6
7     public static void mai
```

Compile time error

The abstract method m in type Test can only be defined by an abstract class
2 quick fixes available:
Remove 'abstract' modifier
Make type 'Test' abstract
suggestions to remove compile time error.
Press 'F2' for focus

```
Console   Debug Shell   TestNG
<terminated> Test (2) [Java Application] C:
```

Note: Abstract class can contain zero no of abstract method or any number of non-abstract method. Happily, we can declare the class as abstract.

Blank.java | Test.java

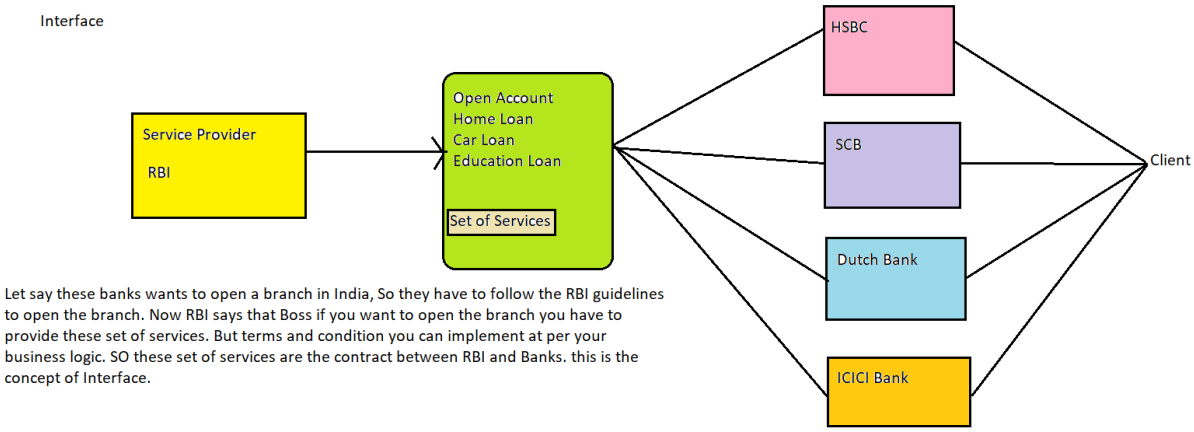
```java
package abstractConceptInterface;

public abstract class Test {

    public void m() {}
    public void m1() {}
    public void m2() {}
    public void m3() {}

    public static void main(String[] args) {
        System.out.println("Abstract class with Non abstract methods");
```

Console | Debug Shell | TestNG

```
<terminated> Test (2) [Java Application] C:\Program Files\Java\jre1.8.0_291\bin\javaw.exe (19-Aug-2022, 4
Abstract class with Non abstract methods
```

 # Interface: Interface in java is a way to achieve abstraction. All the methods in interface by default abstract. It can have only method declaration not body.

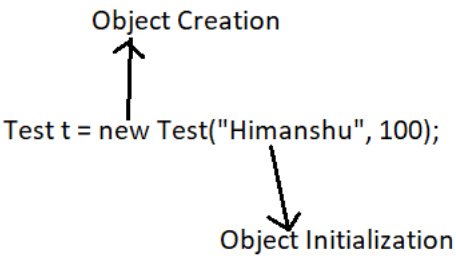
- **➔ Any service requirement specification is called interface**
- **➔ Any contract between client and service provider is called Interface.**



Since Java 8, we can have default and static methods in an interface.

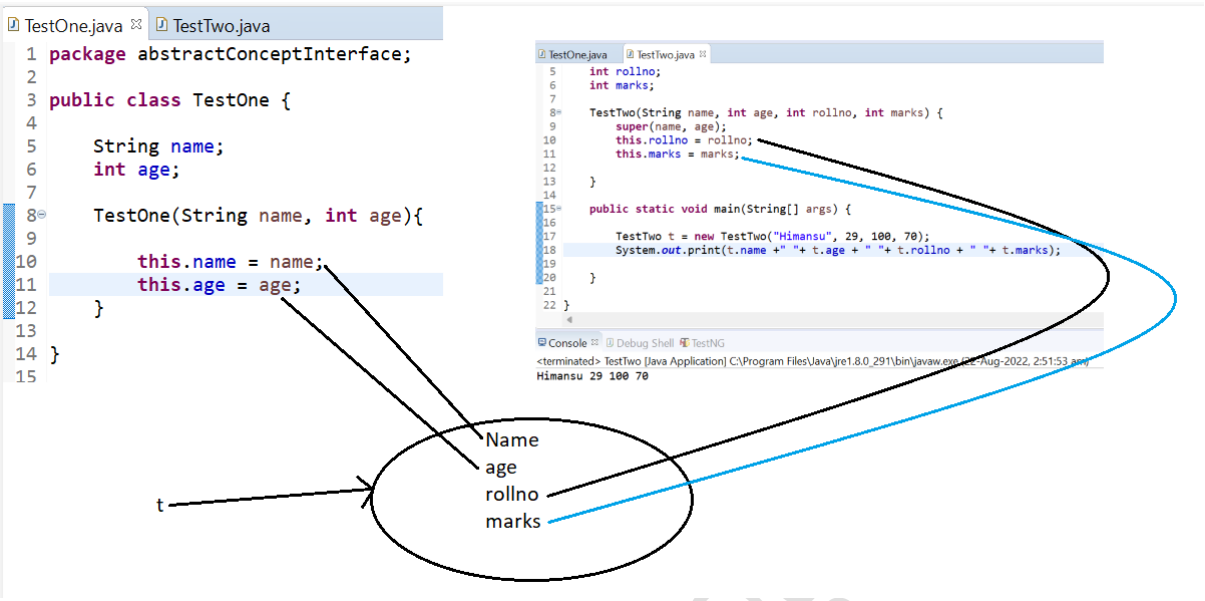
Since Java 9, we can have private methods also in an interface.

Abstract Class & Interface Loopholes:



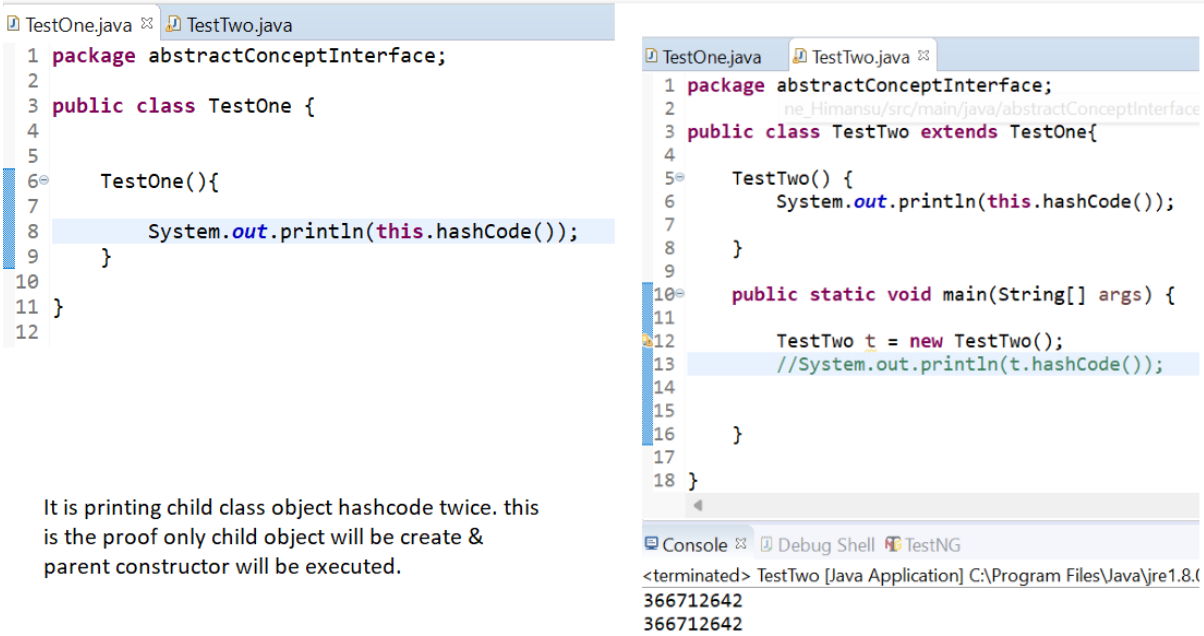
1.

2. **Child Object vs Parent Constructor: Whenever we are creating child class object then automatically parent constructor will be executed to perform initialization for the instance variables which are coming(inheriting) from parent class.**



3. **Whenever we are creating child class object then parent class object will be created or not ?**

No



It is printing child class object hashcode twice. this is the proof only child object will be create & parent constructor will be executed.

Both parent and child class constructor executed for child object only.

4. **Anyway, we can't create object for abstract class either directly or indirectly but abstract class can contain constructor what is the need?**

   The main objective for abstract class constructor is to perform initialization for instance variables which are inheriting from abstract class to the child class. Whenever we are

creating child class object automatically abstract class constructor will be executed to perform initialization for the instance variables which are inheriting from abstract class.

(Code reusability)

TestOne.java | TestTwo.java

```java
package abstractConceptInterface;

public abstract class TestOne {
    int age;
    String name;
    int weight;

    TestOne(int age, String name, int weight){

        this.age = age;
        this.name = name;
        this.weight = weight;
    }

}
```

TestOne.java | TestTwo.java

```java
package abstractConceptInterface;

public class TestTwo extends TestOne{
    int rollno;
    int id;
    TestTwo(int age, String name, int weight, int rollno, int id) {
        super(age, name, weight);
        this.rollno = rollno;
        this.id = id;

    }

    public static void main(String[] args) {

        TestTwo t = new TestTwo(27,"Himansu",72,101, 1001);
        System.out.println(t.name + " " + t.age + " " + t.weight + " " + t.rollno + " " + t.id);
```

Console | Debug Shell | TestNG

```
<terminated> TestTwo [Java Application] C:\Program Files\Java\jre1.8.0_291\bin\javaw.exe (22-Aug-2022, 3:51:13 am)
Himansu 27 72 101 1001
```

5. **Abstract class contain instance variables that's why constructor concept is application to perform initialization. But in Interface every variable is by default Public, Static & Final so there is no existence of instance variable in Interface that's why constructor concept is not applicable for Interface.**
6. **If everything is abstract then highly recommended to go with interface but not with abstract class.**
7. **We can replace interface with abstract class but it is not good programming practice.**
8. **While implementing interface we can extends any other class and hence we won't miss inheritance benefit. While extending abstract class we can't extend any other class and hence we are missing inheritance benefit.**

Interface	Abstract Class
1. If we don't know anything about implementation just we have requirement specification then we should go for interface.	1. If we are talking about implementation but not completely (partial implementation) then we should go for Abstract class.
2. Inside Interface every method is always public and abstract whether we are declaring or not. Hence interface is also considered as 100% pure Abstract class.	2. Every method present in Abstract class need not be public and Abstract. In addition to abstract methods we can take concrete methods also.
6. For Interface variables compulsory we should perform initialization at the time of declaration otherwise we will get compile time error.	6. For Abstract class variables it is not required to perform initialization at the time of declaration.
7. Inside interface we can't declare instance and static blocks. Otherwise we will get compile time error.	7. Inside Abstract class we can declare instance and static blocks.
4. Every variable present inside interface is always public, static and final whether we are declaring or not.	4. The variables present inside Abstract class need not be public static and final.

1. **Whenever we are implementing interface method, it is compulsory to declare that method as Public.**
2. **Whenever we are implementing interface, for each and every abstract method of that interface we should provide implementation. If you are unable to provide the implementation for at least one method then declare that class as abstract. Then who will provide the implementation for that remaining one method, that is next child class. Otherwise we will get compile time error.**

```
Person.java   PersonInterface.java
1 package abstractConceptInterface;
2
3 public class PersonInterface implements Person{
4
5     public v
6
7
8     }
9
10 }
11
```

The type PersonInterface must implement the inherited abstract method Person.m2()

2 quick fixes available:

- Add unimplemented methods
- Make type 'PersonInterface' abstract

Press 'F2' for focus

```
Person.java   PersonInterface.java
1 package abstractConceptInterface;
2
3 public interface Person {
4
5     void m1();
6     void m2();
7
8 }
9
```

Now we can implement the other method m2() and can resolve the problem.

```
Person.java   *PersonInterface.java
1 package abstractConceptInterface;
2
3 public class PersonInterface implements Person{
4
5     public void m1() {
6
7
8     }
9
10    public void m2() {
11
12    }
13
14 }
15
```

Or we can declare the class as an Abstract and can resolve the problem.

```
Person.java   PersonInterface.java
1 package abstractConceptInterface;
2
3 public abstract class PersonInterface implements Person{
4
5     public void m1() {
6
7
8     }
9
10
11 }
12
```

Now we can see that compile time error is gone by declaring the class as abstract.

Now who is responsible to implement method m2(). We can see that we are getting compile time error because we are extending ChildInterfaceClass.

```
Person.java    PersonInterface.java    ChildInterfaceClass.java
1 package abstractConceptInterface;
2
3 public class ChildInterfaceClass extends PersonInterface{
4
5 }
6
```

The type ChildInterfaceClass must implement the inherited abstract method Person.m2()

2 quick fixes available:

- Add unimplemented methods
- Make type 'ChildInterfaceClass' abstract

Press 'F2' for focus

Now again we can make it as abstract or we can implement the method m2() here. And error will go.

```
Person.java    PersonInterface.java    ChildInterfaceClass.java
1 package abstractConceptInterface;
2
3 public class ChildInterfaceClass extends PersonInterface{
4
5     public void m2() {
6
7     }
8
9 }
10
```

If there are abstract methods present in abstract class then also you need to provide the implementation in child class otherwise you will get the compile time error.

```
Person.java    PersonInterface.java    ChildInterfaceClass.java
1 package abstractConceptInterface;
2
3 public class ChildInterfaceClass extends PersonInterface{
4
5     public vo
6
7     }
8
9 }
10
```

The type ChildInterfaceClass must implement the inherited abstract method PersonInterface.m3()

2 quick fixes available:

- Add unimplemented methods
- Make type 'ChildInterfaceClass' abstract

Press 'F2' for focus

SO again we are getting 2 options to resolve the problem one is we can implement the m3() method here or we can make the class as an abstract.

Person.java | PersonInterface.java | ChildInterfaceClass.java

```java
package abstractConceptInterface;

public class ChildInterfaceClass extends PersonInterface{

    public void m2() {

    }

    @Override
    void m3() {


    }

}
```

Now you can see that error is gone by implementing the method m3(). Or we could choose another option as well by declaring class as abstract.