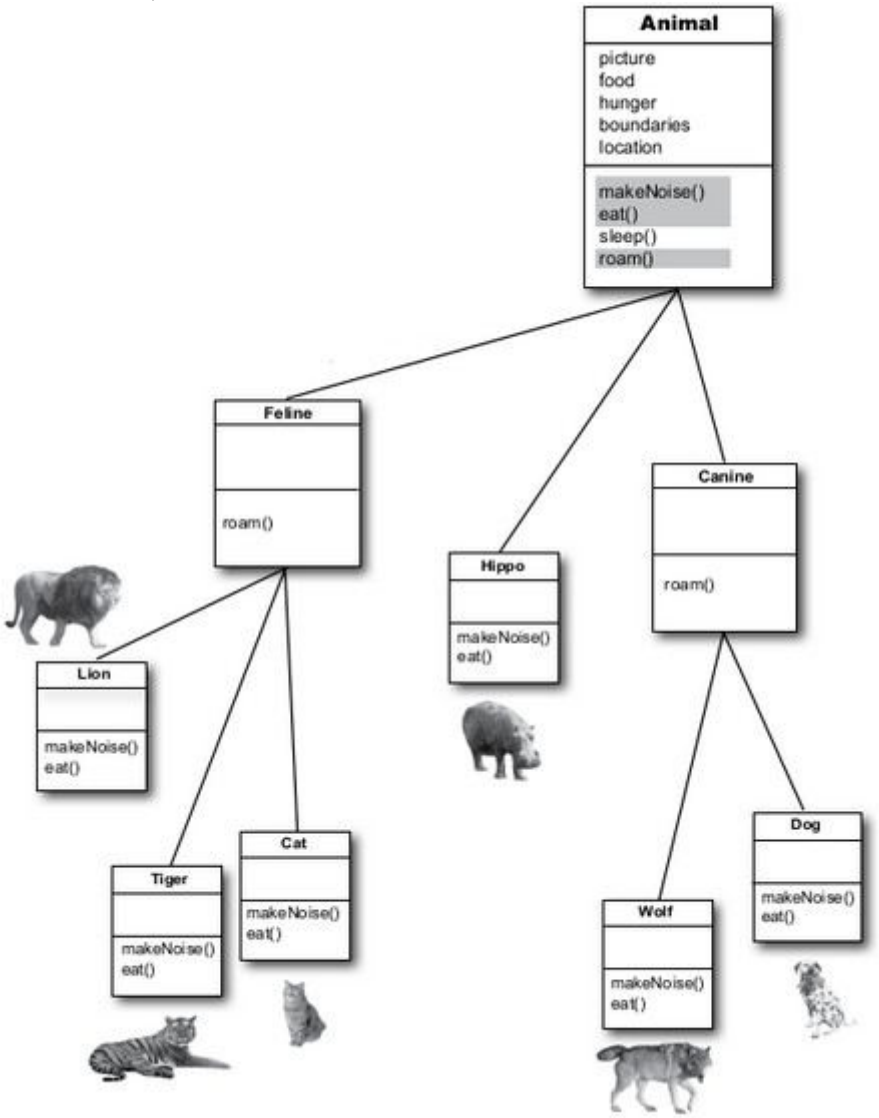
Chapter-3

Inheritance and Interface

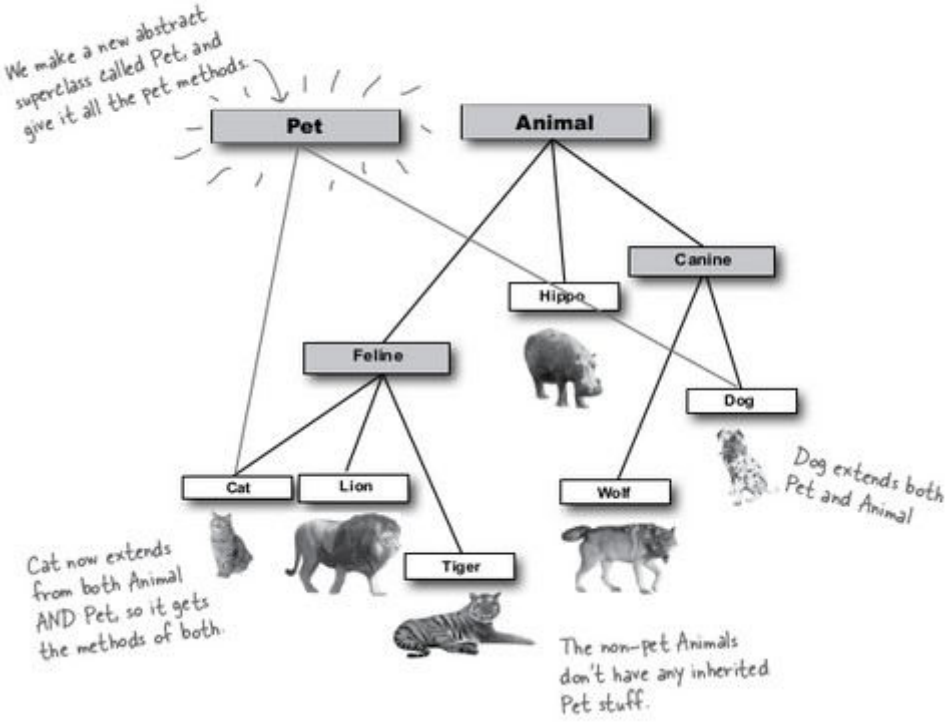
Index

Inheritance:

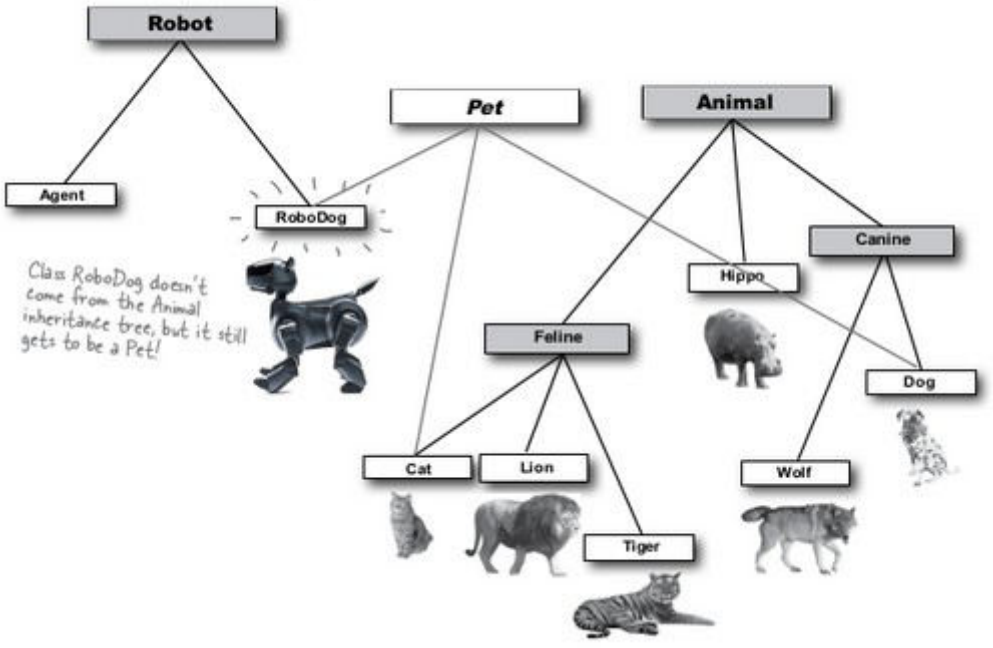
Interfaces:



It looks like we need TWO superclasses at the top



Classes from *different* inheritance trees can implement the *same* interface.



INHERITANCE

Agenda

1. Inheritance(IS-A relationship)
2. Types of inheritance:
   - Single inheritance
   - Multilevel inheritance
   - Hierarchical inheritance
   - Multiple inheritance
   - Hybrid inheritance
   - Cyclic inheritance

Inheritance(IS-A relationship):-

1. The process of acquiring fields (variables) and methods(behaviors) from one class to another class is called inheritance.
2. The main objective of inheritance is code extensibility, whenever we are extending class automatically the code is reused.
3. In inheritance, one class giving the properties and behavior & another class is taking the properties and behavior.
4. Inheritance is also known as **is-a** relationship. By using extends keyword we are achieving inheritance concept.
5. extends keyword used to achieve inheritance & it is providing relationship between two classes. when you make relationship then able to reuse the code.
6. In java, parent class is giving properties to child class and Child is acquiring properties from Parent.
7. To reduce length of the code and redundancy of the code sun people introduced inheritance concept.

Application code before inheritance	Application code after inheritance
class A ***{ void m1(){ }*** ***void m2(){ }*** ***}*** ***class B*** ***{ void m1(){ }*** ***void m2(){ }*** ***void m3(){ }*** ***void m4(){ }*** ***}*** ***class C*** ***{ void m1(){ }*** ***void m2(){ }*** ***void m3(){ }*** ***void m4(){ }*** ***void m5(){ }*** ***void m6(){ }*** ***}***	***class A //parent class or super class or base*** ***{*** ***void m1(){ }*** ***void m2(){ }*** ***}*** ***class B extends A //child or sub or derived*** ***{*** ***void m3(){ }*** ***void m4(){ }*** ***}*** ***class C extends B*** ***{*** ***void m5(){ }*** ***void m6(){ }*** ***}***

Note 1:-In java it is possible to create objects for both parent and child classes.

1. If we are creating object for parent class it is possible to call only parent specific methods.

```
A a=new A();
a.m1();
a.m2();
```

2. If we are creating object for child class it is possible to call parent specific and child specific.

```
B b=new B();
b.m1();
b.m2();
b.m3();
b.m4();


C c=new C();
c.m1();
c.m2();
c.m3();
c.m4();
c.m5();
c.m6();
```

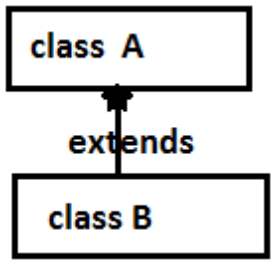
Types of inheritance :-

There are six types of inheritance.

1) Single inheritance
2) Multilevel inheritance
3) Hierarchical inheritance
4) Multiple inheritance
5) Hybrid inheritance
6) Cyclic inheritance

1) Single inheritance:-

- One class has one and only one direct super class is called single inheritance.
- In the absence of any other explicit super class, every class is implicitly a subclass of Object class.



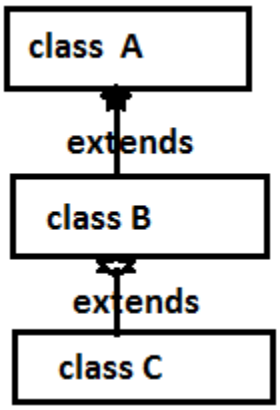
Class B extends A===>class B acquiring properties of A class.

Example:-

```
class Parent
{       void property()
        {
                System.out.println("money");
        }
}
class Child extends Parent
{       void m1()
        {
                System.out.println("m1 method");
        }
        public static void main(String[ ] args)
        {
                Child c = new Child();
                c.property();     //parent class method executed
                c.m1();           //child class method executed
        }
}
```

2) Multilevel inheritance:-

One Sub class is extending Parent class then that sub class will become Parent class of next extended class this flow is called multilevel inheritance.



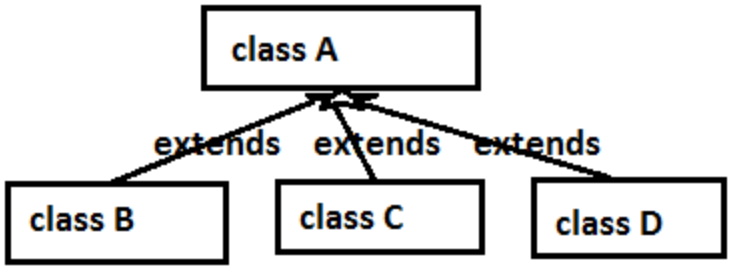
Class B extends A ***===> class B acquiring properties of A***
class Class C extends B ***===> class C acquiring properties of B class***
[indirectly class C using properties of A & B classes]

Example:-

```
class A
{       void m1()
        {
                System.out.println("m1 method");
        }
}
class B extends A
{       void m2()
        {
                System.out.println("m2 method");
        }
}
class C extends B
{       void m3()
        {
                System.out.println("m3 method");
        }
         public static void main(String[] args)
        {       A a = new A();          a.m1();
                B b = new B();          b.m1(); b.m2();;
                C c = new C();          c.m1(); c.m2(); c.m3();
        }
}
```

3) Hierarchical inheritance :-

More than one sub class is extending single Parent is called hierarchical inheritance.



Class B extends A ===> class B acquiring properties of A class
Class C extends A ===>class C acquiring properties of A class
Class D extends A ===>class D acquiring properties of A class

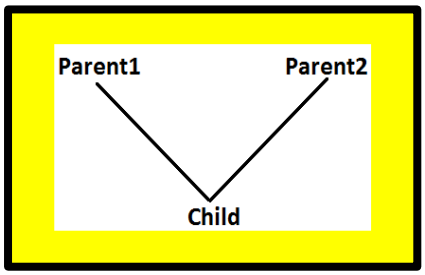
Example:-

```
class A
{       void m1()
        {
                System.out.println("A class");
        }
}
class B extends A
{       void m2()
        {
                System.out.println("B class");
        }
}
class C extends A
{       void m2()
        {
                System.out.println("C class");
        }
}
class Test
{       public static void main(String[] args)
        {       B b= new B();
                b.m1();
                b.m2();
                C c = new C();
                c.m1();
                c.m2();
        }
}
```

4) Multiple inheritance:-

Having more than one Parent class at the same level is called multiple inheritance.

Example:

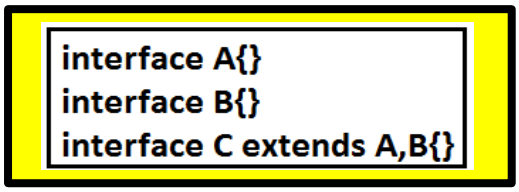


Any class can extends only one class at a time and can't extends more than one class Simultaneously. Hence java won't provide support for multiple inheritance.

Example:

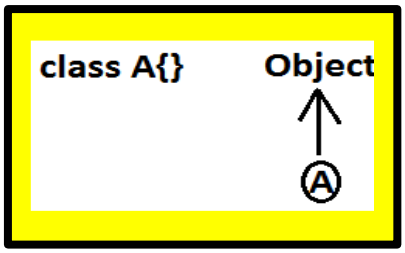
But an interface can extends any no. of interfaces at a time. Hence java provides support for multiple inheritance through interfaces.

Example:



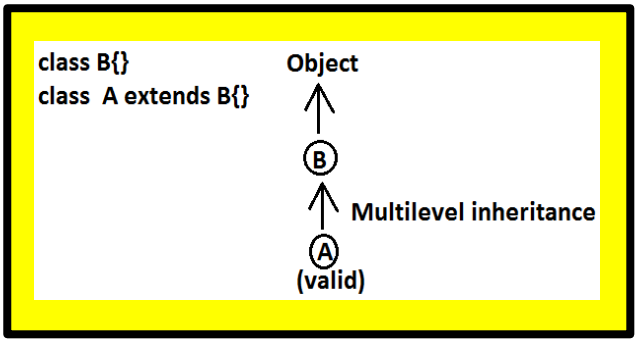
If our class doesn't extends any other class then only our class is the direct child class of object.

Example:

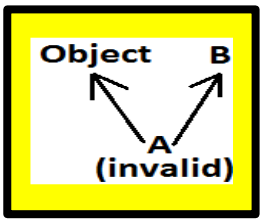


If our class extends any other class then our class is not direct child class of object, It is indirect child class of object , which forms multilevel inheritance.

Example 1:



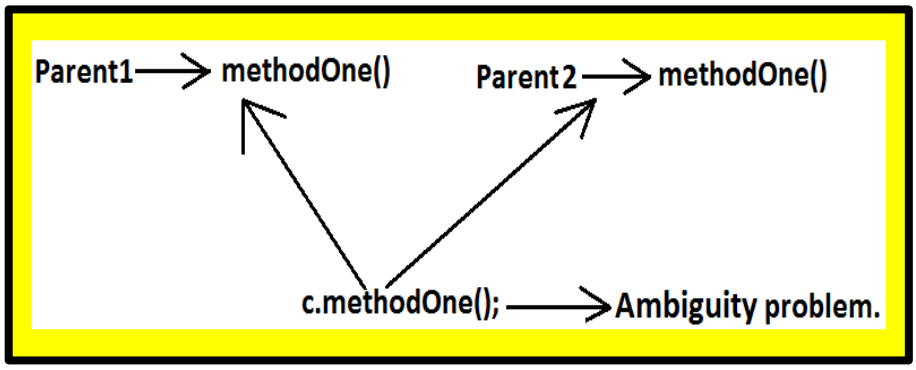
Example 2:



Why java won't provide support for multiple inheritance?

There may be a chance of raising ambiguity problems.

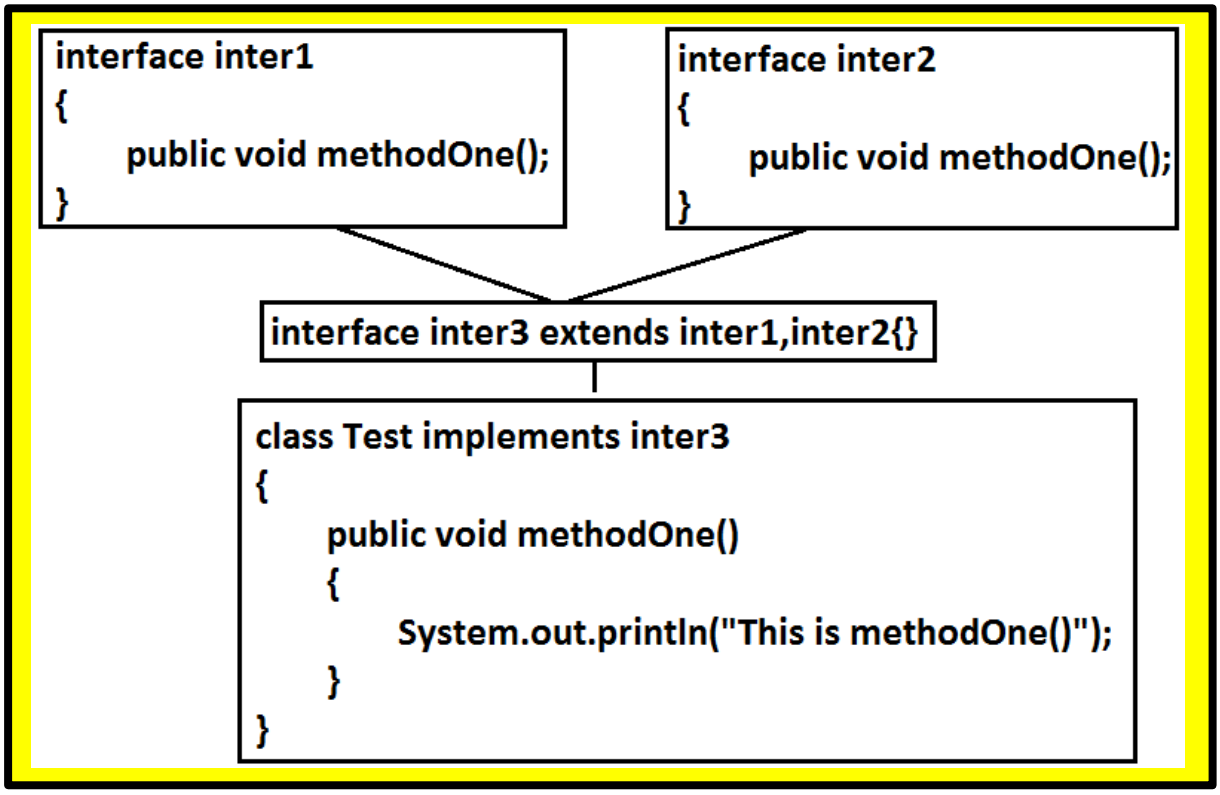
Example:



Why ambiguity problem won't be there in interfaces?

Interfaces having dummy declarations and they won't have implementations hence no ambiguity problem.

Example:



Example program for multiple inheritance:

```
interface Father
{
        float HT=6.2F;
        void height();
}
interface Mother
{
        float HT=5.8F;
        void height();
}
class child implements Father,Mother
{
        public void height()
        {
                float ht=(Father.HT+Mother.HT)/2;
                System.out.println("child's height="+ht);
        }
}
class Multi
{
        public static void main(String[] args)
        {
                child ch=new child();
                ch.height();
        }
}
```

Output:

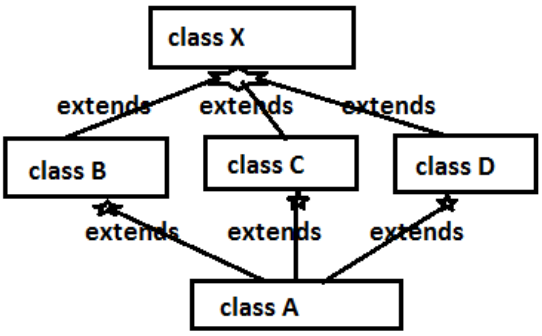
$ javac Multi.java

$ java Multi

child's height=6.0

5) Hybrid inheritance:-

- ➢ Hybrid is combination of hierarchical & multiple inheritance .
- ➢ Java is not supporting hybrid inheritance because multiple inheritance(not supported by java) is included in hybrid inheritance.

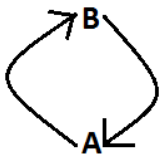


6) Cyclic inheritance:

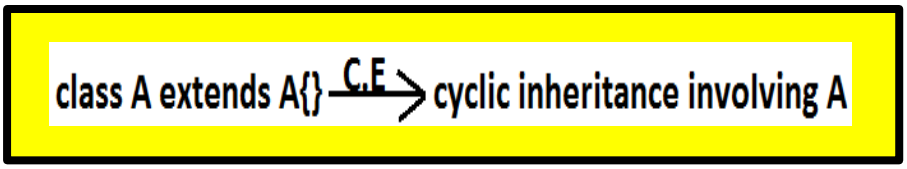
Cyclic inheritance is not allowed in java.

Example 1:

class A extends B{} } **(invalid)**
class B extends A{} } **C.E:cyclic inheritance involving A**



Example 2:



Preventing inheritance:-

- You can prevent sub class creation by using final keyword in the parent class declaration.

```
final class Parent //for this class child class creation not possible because it is final.
{
}
class Child extends Parent
{
}
```

compilation error:- cannot inherit from final Parent

Note:-

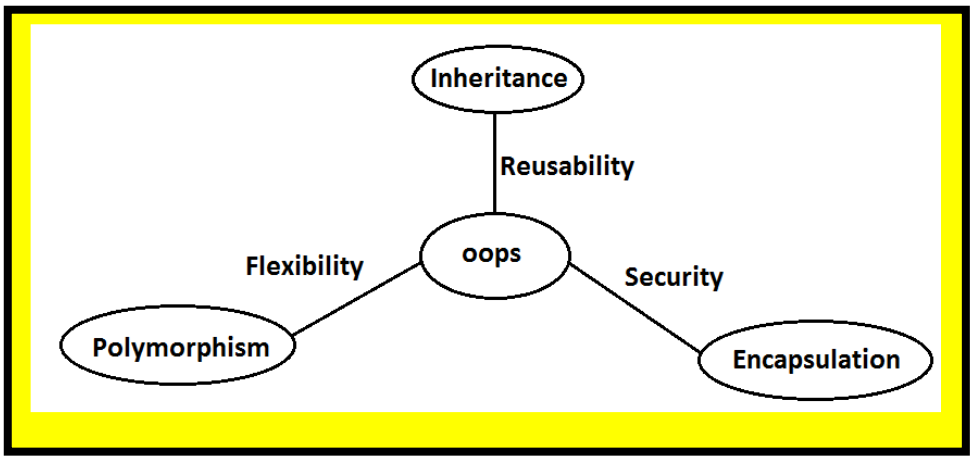
1) Except for the Object class , a class has one direct super class.
2) A class inherit fields and methods from all its super classes whether directly or indirectly.
3) An abstract class can only be sub classed but cannot be instantiated.
4) In parent and child, it is recommended to create object of Child class.

Polymorphism

Agenda:

1. **Polymorphism**
   a) **Overloading**
      - **Automatic promotion in overloading**

Pillars of OOPS:



1) Inheritance talks about reusability.
2) Polymorphism talks about flexibility.
3) Encapsulation talks about security.

Polymorphism:

- Polymorphism is a Greek word poly means many and morphism means forms.
- Same name with different forms is the concept of polymorphism.

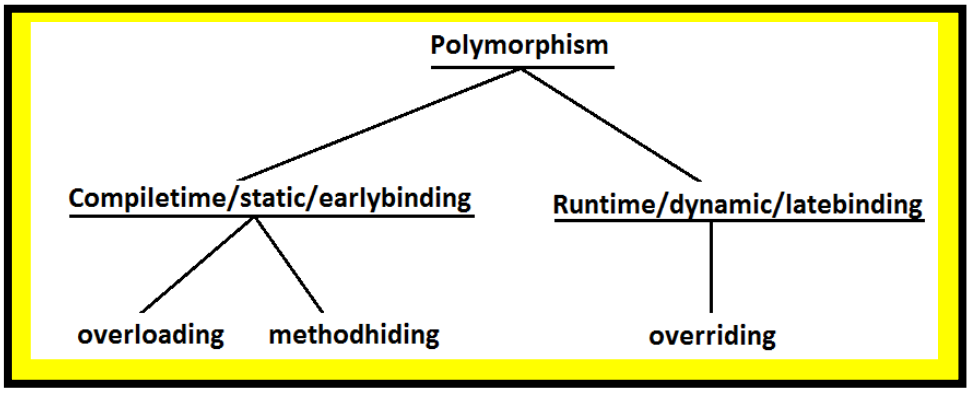
Ex 1: We can use **same abs( ) method** for int type, long type, float type etc.
Ex:

1. abs(int)
2. abs(long)
3. abs(float)

Ex 2: We can use the **parent reference** to hold any **child objects**. We can use the same List reference to hold ArrayList object, LinkedList object, Vector object, or Stack object.
Ex:

1. List l=new ArrayList( );
2. List l=new LinkedList( );
3. List l=new Vector( );
4. List l=new Stack( );

Diagram:

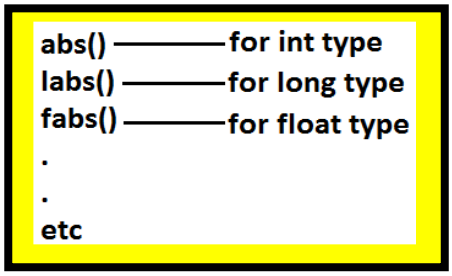


There are two types of polymorphism in java

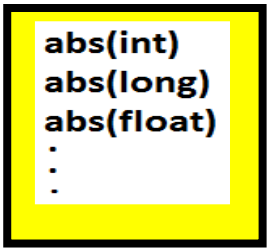
1. **Compile time polymorphism**: Its method execution decided at compilation time.
   Example :- method overloading.
2. **Runtime polymorphism**: Its method execution decided at runtime.
   Example :- method overriding.

Overloading:-

1) Two methods are said to be overload, if and only if both having the same name but different argument types.
2) In 'C' language, we can't take 2 methods with the same name and different types. If there is a change in argument type, compulsory we should go for new method name.
   Example:



3) Lack of overloading in "C" increases complexity of the programming.
4) But in java, we can take multiple methods with the same name and different argument types.
   Example:



5) Having the same name and different argument types is called method overloading.

6) All these methods are considered as overloaded methods.
7) Having overloading concept in java reduces complexity of the programming.

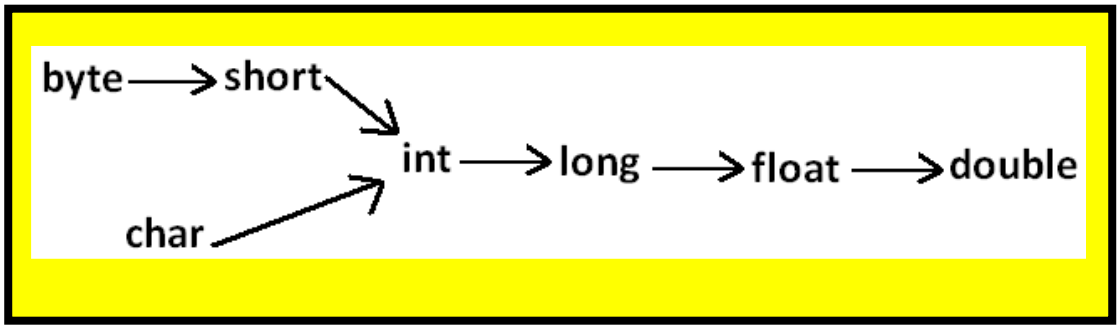
```
class Test
{
        public void methodOne()
        {
                System.out.println("no-arg method");
        }
        public void methodOne(int i)
        {
                System.out.println("int-arg method");
           //overloaded methods
        }
        public void methodOne(double d)
        {
                System.out.println("double-arg method");
        }
        public static void main(String[] args)
        {
                Test t=new Test();
                t.methodOne();        //no-arg method
                t.methodOne(10);      //int-arg method
                t.methodOne(10.5);//double-arg method
        }
}
```

Conclusion : In overloading, compiler is responsible to perform method resolution(decision) based on the reference type(but not based on run time object). Hence overloading is also considered as compile time polymorphism(or) static polymorphism (or)early biding.

Case 1: Automatic promotion in overloading.

- In overloading if compiler is unable to find the method with exact match we won't get any compile time error immediately.
- First compiler promotes the argument to the next level and checks whether the matched method is available or not. If it is available, then that method will be considered. If it is not available, then compiler promotes the argument once again to the next level. This process will be continued until all possible promotions still if the matched method is not available then we will get compile time error. This process is called automatic promotion in overloading.

The following are various possible automatic promotions in overloading.



Example:

```
class Test
{
	public void methodOne(int i)
	{
		System.out.println("int-arg method");
	}
	public void methodOne(float f) //overloaded methods
	{
		System.out.println("float-arg method");
	}
	public static void main(String[] args)
	{
		Test t=new Test();
		//t.methodOne('a');//int-arg method
		//t.methodOne(10l);//float-arg method
		t.methodOne(10.5);//C.E:cannot find symbol
	}
}
```

Overriding

Overriding
- **Rules for overriding**
- **Overriding with Checked Exceptions**
- **Overriding with respect to static methods.**

Method hiding
- **Differences between method hiding and overriding**

Overriding :

1. Whatever the Parent has by default available to the Child through inheritance, if the Child is not satisfied with Parent class method implementation then Child is allow to redefine that Parent class method in Child class in its own way this process is called overriding.
2. The Parent class method which is overridden is called overridden method.
3. The Child class method which is overriding is called overriding method.

```
Example 1:
class Parent
{
        public void property()
        {
                System.out.println("cash+land+gold");
        }
        public void marry()
        {
                System.out.println("subbalakshmi");              //overridden method
        }
}
class Child extends Parent
{
        public void marry()
        {
                System.out.println("3sha/4me/9tara/anushka"); //overriding method
        }
}
class Test
{
        public static void main(String[] args)
        {
                Parent p=new Parent();
                p.marry();                      //subbalakshmi(parent method)
                Child c=new Child();
                c.marry();                      //3sha/4me/9tara(child method)
                Parent p1=new Child();
```

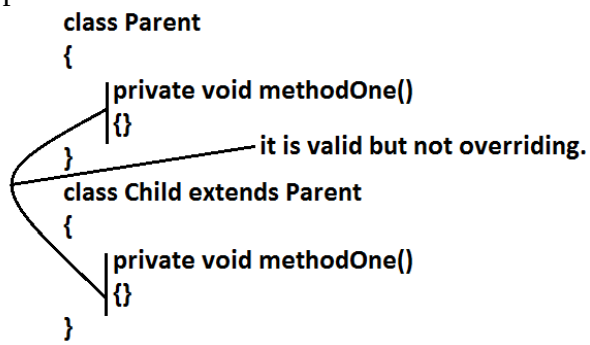
```
                p1.marry();                    //3sha/4me/9tara(child method)
        }
}
```

4. In overriding method resolution is always takes care by JVM based on runtime object hence overriding is also considered as runtime polymorphism or dynamic polymorphism or late binding.
5. The process of overriding method resolution is also known as dynamic method dispatch.

Note: In overriding runtime object will play the role and reference type is dummy.

Rules for overriding :

1. In overriding method names and arguments must be same. That is method signature must be same.
2. Private methods are not visible in the Child classes, hence overriding concept is not applicable for private methods. Based on own requirement we can declare the same Parent class private method in child class also. It is valid but not overriding.
   Example:



3. Parent class final methods we can't override in the Child class.

```
Example:
class Parent
{
        public final void methodOne()          {}
}
class Child extends Parent{
        public void methodOne(){}
}
Output:
Compile time error:
 methodOne() in Child cannot override methodOne()
    in Parent; overridden method is final
```

Parent class non final methods we can override as final in child class. We can override native methods in the child classes.

4. We should override Parent class abstract methods in Child classes to provide implementation.

```
Example:
abstract class Parent
{
        public abstract void methodOne();
}
class Child extends Parent
{
        public void methodOne() { }
}
```

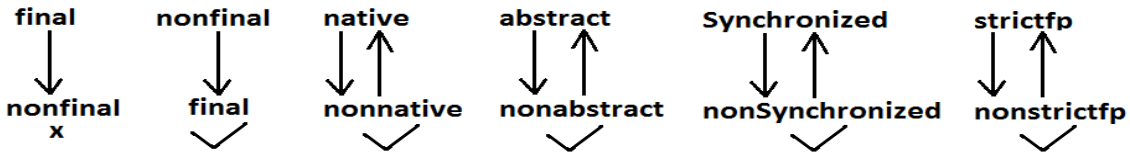
Diagram:

5. We can override a non-abstract method as abstract
   this approach is helpful to stop availability of Parent method implementation to the next level child classes.

```
Example:
class Parent
{
    public void methodOne()  { }
}
abstract class Child extends Parent
{
    public abstract void methodOne();
}
```

Synchronized, strictfp, modifiers won't keep any restrictions on overriding.

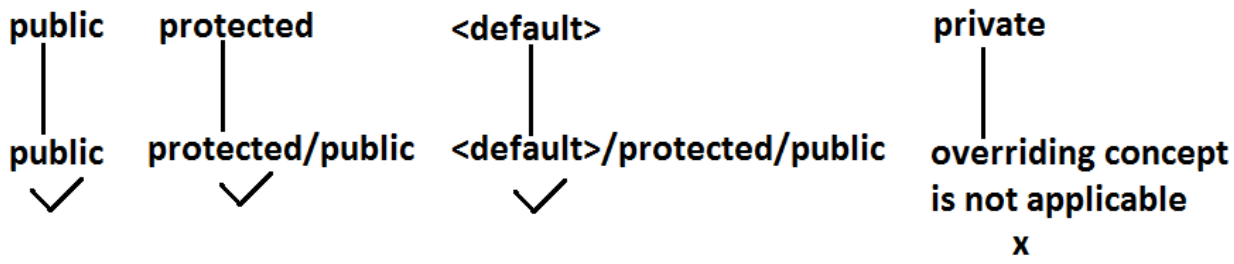
Diagram:



6. While overriding we can't reduce the scope of access modifier.

```
Example:
class Parent
{
    public void methodOne() { }
}
class Child extends Parent
{
    protected void methodOne( )  { }
}
Output:
Compile time error :
methodOne() in Child cannot override methodOne() in Parent;
 attempting to assign weaker access privileges; was public
```

Diagram:

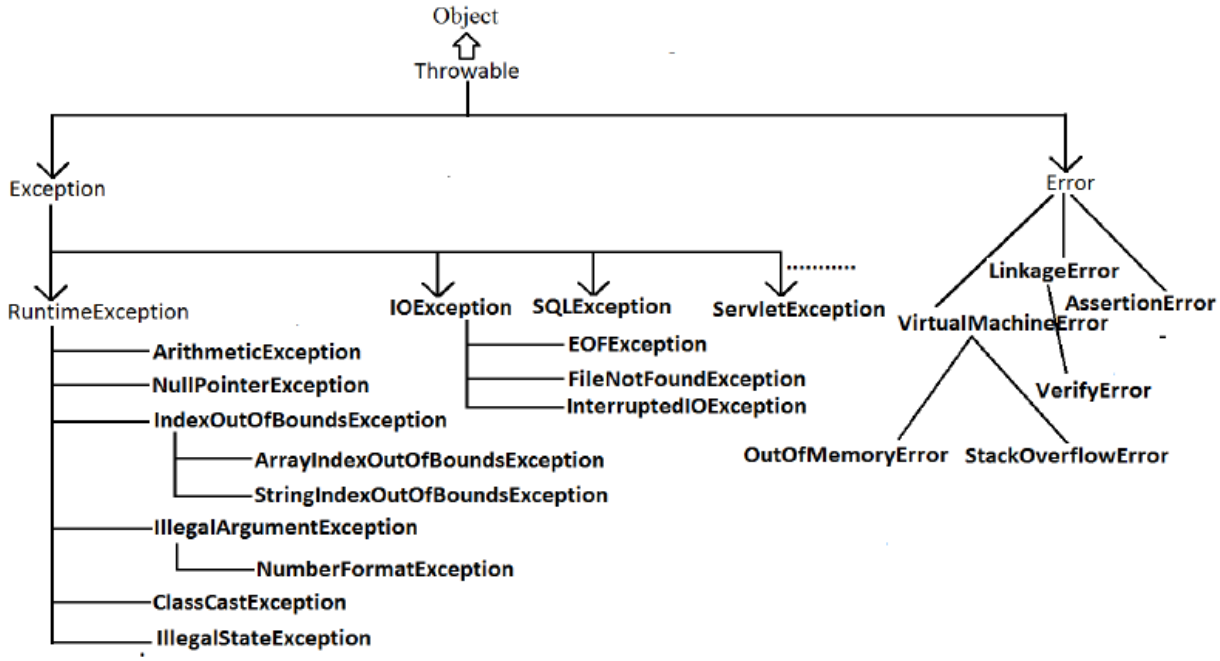


private < default < protected < public

Overriding with Checked Exceptions :

- The exceptions which are checked by the compiler for smooth execution of the program at runtime are called checked exceptions.
- The exceptions which are not checked by the compiler are called un-checked exceptions.
- RuntimeException and its child classes, Error and its child classes are unchecked except these the remaining are checked exceptions.

Diagram:



Rule: While overriding if the child class method throws any checked exception, compulsory the parent class method should throw the same checked exception or its parent, otherwise we will get compile time error.

But there are no restrictions for un-checked exceptions.

Example:

```
class Parent
{
    public void methodOne() {}
}
class Child extends Parent
{
    public void methodOne() throws Exception {}
}
```

Output:

Compile time error :

methodOne() in Child cannot override methodOne() in Parent;
overridden method does not throw java.lang.Exception

Examples :

① Parent: public void methodOne()throws Exception
Child: public void methodOne()
} valid

② Parent: public void methodOne()
Child : public void methodOne()throws Exception
} invalid

③ Parent: public void methodOne()throws Exception
Child: public void methodOne()throws Exception
} valid

④ Parent: public void methodOne()throws IOException
Child: public void methodOne()throws Exception
} invalid

⑤ Parent: public void methodOne()throws IOException
Child: public void methodOne()throws EOFException,FileNotFoundException
} valid

⑥ Parent: public void methodOne()throws IOException
Child : public void methodOne()throws EOFException,InterruptedException
} invalid

⑦ Parent: public void methodOne()throws IOException
Child: public void methodOne()throws EOFException,ArithmeticException
} valid

⑧ Parent: public void methodOne()
Child: public void methodOne()throws
ArithmeticException,NullPointerException,ClassCastException,RuntimeException
} valid

Overriding with respect to static methods:

Case 1:

We can't override a static method as non static.

Example:

```
class Parent
{
        public static void methodOne(){}
  //here static methodOne() method is a class level
}
class Child extends Parent
{
        public void methodOne(){}
//here methodOne() method is a object level hence
  // we can't override methodOne() method
}
```

output :

```
CE: methodOne in Child  can't  override  methodOne() in Parent ;
  overriden method is  static
```

Case 2:

Similarly we can't override a non static method as static.

Case 3:

```
class Parent
{
    public static void methodOne() {}
}
class Child extends Parent {
    public static void methodOne() {}
}
```

It is valid. It seems to be overriding concept is applicable for static methods but it is not overriding it is method hiding.

METHOD HIDING:

All rules of method hiding are exactly same as overriding except the following differences.

Overriding	Method hiding
1. Both Parent and Child class methods should be non static.	1. Both Parent and Child class methods should be static.
2. Method resolution is always takes care by JVM based on runtime object.	2. Method resolution is always takes care by compiler based on reference type.
3. Overriding is also considered as runtime polymorphism (or) dynamic polymorphism (or) late binding.	3. Method hiding is also considered as compile time polymorphism (or) static polymorphism (or) early binding.

Example:

```
class Parent
{
        public static void methodOne()
        {
                System.out.println("parent class");
        }
}
class Child extends Parent
{
        public static void methodOne()
        {
                System.out.println("child class");
        }
}
```

```
class Test
{
	public static void main(String[] args)
	{
		Parent p=new Parent();
		p.methodOne();		//parent class
		Child c=new Child();
		c.methodOne();		//child class
		Parent p1=new Child();
		p1.methodOne();		//parent class
	}
}
```

Note: If both Parent and Child class methods are non static then it will become overriding and method resolution is based on runtime object. In this case the output is

```
Parent class
Child class
Child class
```

Dynamic method dispatch in java

Dynamic method dispatch is a mechanism to resolve overridden method call at run time instead of compile time. It is based on the concept of up-casting (A super class reference variable can refer subclass object.).

Example 1:

```
/**
 * This program is used for simple method overriding example.
 * with dynamic method dispatch.
 */
class Student {
	/**
	 * This method is used to show details of a student.
	 */
	public void show(){
		System.out.println("Student details.");
	}
}

public class CollegeStudent extends Student {
	/**
	 * This method is used to show details of a college student.
	 */
	public void show(){
		System.out.println("College Student details.");
	}

	//main method
	public static void main(String args[]){
	 //Super class can contain subclass object.
	 Student obj = new CollegeStudent();

	 //method call resolved at runtime
	 obj.show();
	}
}
```

Output:

```
College Student details.
School Student details.
```

Note: Only super class methods can be overridden in subclass, data members of super class cannot be overridden.

```java
/**
 * This program is used for simple method overriding example.
 * with dynamic method dispatch.
 */
class Student {
      int maxRollNo = 200;
}

class SchoolStudent extends Student{
      int maxRollNo = 120;
}

class CollegeStudent extends SchoolStudent{
      int maxRollNo = 100;
}

public class StudentTest {
      public static void main(String args[]){
       //Super class can contain subclass object.
       Student obj1 = new CollegeStudent();
       Student obj2 = new SchoolStudent();

       //In both calls maxRollNo of super class will be printed.
       System.out.println(obj1.maxRollNo);
       System.out.println(obj2.maxRollNo);
      }
}
```

Output:

```
200
200
```

Final modifier

Agenda:

1. Final Modifier
    a. Final Methods
    b. Final Class
    c. Final Variables
        i. Final instance variables
            - At the time of declaration
            - Inside instance block
            - Inside constructor
        ii. Final static variables
            - At the time of declaration
            - Inside static block
        iii. Final local variables

Final modifier:

Final is the modifier applicable for classes, methods and variables.

a) Final methods:

- Whatever methods parent has, by default available to the child.
- If the child is not allowed to override any method, that method we have to declare with final in parent class i.e., final methods cannot be overridden.

Example:

Program 1:

```
class Parent
{
        public void property()
        {
                System.out.println("cash+gold+land");
        }
        public final void marriage()
        {
                System.out.println("Venkata laxmi");
        }
}
class child extends Parent
{
        public void marriage()
        {
                System.out.println("Samantha");
        }
}
```

OUTPUT:

```
Compile time error.

D:\Java>javac child.java
child.java:3: marriage() in child cannot override marriage() in Parent;
overridden method is final
```

b) Final class:

If a class declared as final, then we can't create the child class i.e., inheritance concept is not applicable for final classes.

Example:

Program 1:

```
final class Parent
{
}
class child extends Parent
{
}
```

OUTPUT:

```
Compile time error.

D:\Java>javac child.java
child.java:1: cannot inherit from final Parent
class child extends Parent
```

Note: Every method present inside a final class is always final by default, whether we are declaring or not. But every variable present inside a final class need not be final.

Example:

```
final class parent
{
        static int x=10;
        static
        {
                x=999;
        }
}
```

Advantages and disadvantages of final keyword:

Advantage:

we can achieve security.

Disadvantage:

we are missing the key benefits of OOPS:

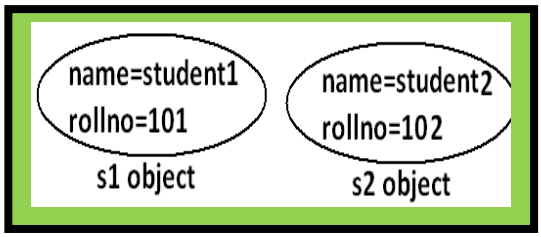
i. polymorphism (because of final methods),
ii. inheritance (because of final classes) hence if there is no specific requirement never recommended to use final keyboard.

c) Final variables:

Final instance variables:

- If the value of a variable is varied from object to object such type of variables are called instance variables.
- For every object a separate copy of instance variables will be created.

DIAGRAM:



For the instance variables it is not required to perform initialization explicitly jvm will always provide default values.

Example:

```
class Test
{
	int i;
	public static void main(String args[])
	{
		Test t=new Test();
		System.out.println(t.i);
	}
}
Output:
D:\Java>javac Test.java
D:\Java>java Test
0
```

If the instance variable declared as final, compulsory we should perform initialization explicitly and JVM won't provide any default values. whether we are using or not otherwise we will get compile time error.

Example:

Program 1:

```
class Test
{
	int i;
}
Output:
D:\Java>javac Test.java
D:\Java>
```

Program 2:

```
class Test
{
	final int i;
}
```

```
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:1: variable i might not have been initialized
class Test
```

Rule:

For the final instance variables we should perform initialization before constructor completion. That is the following are various possible places for this.

1) At the time of declaration:

Example:

```
class Test
{
	final int i=10;
}
Output:
D:\Java>javac Test.java
D:\Java>
```

2) Inside instance block:

Example:

```
class Test
{
	final int i;
	{
		i=10;
	}
}
Output:
D:\Java>javac Test.java
D:\Java>
```

3) Inside constructor:

Example:

```
class Test
{
	final int i;
	Test()
	{
		i=10;
	}
}
Output:
D:\Java>javac Test.java
D:\Java>
```

If we are performing initialization anywhere else we will get compile time error.

Example:

```
class Test
{
	final int i;
	public void methodOne()
	{
		i=10;
	}
}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:5: cannot assign a value to final variable i
i=10;
```

Final static variables:

- If the value of a variable is not varied from object to object such type of variables is not recommended to declare as the instance variables. We have to declare those variables at class level by using static modifier.

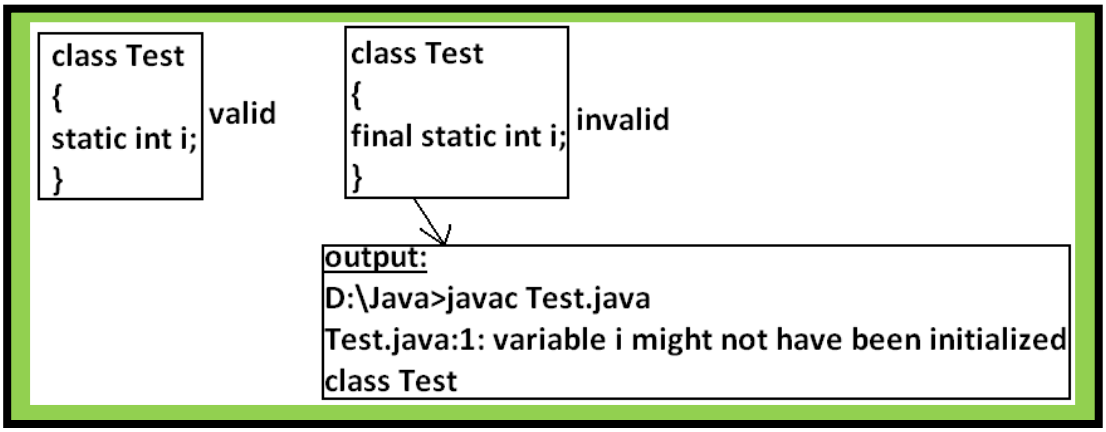
- In the case of instance variables, for every object a seperate copy will be created. But in the case of static variables, a single copy will be created at class level and shared by every object of that class.
- For the static variables it is not required to perform initialization, explicitly jvm will always provide default values.

Example:

```
class Test
{
	static int i;
	public static void main(String args[])
	{
		System.out.println("value of i is :"+i);
	}
}
Output:
D:\Java>javac Test.java
D:\Java>java Test
Value of i is: 0
```

If the static variable declare as final, then compulsory we should perform initialization explicitly whether we are using or not otherwise we will get compile time error.(The JVM won't provide any default values)

Example:



Rule:

For the final static variables we should perform initialization before class loading completion otherwise we will get compile time error. That is the following are possible places.

1) At the time of declaration:

Example:

```
class Test
{
	final static int i=10;
}
Output:
D:\Java>javac Test.java
D:\Java>
```

2) Inside static block:

Example:

```
class Test
{
	final static int i;
	static
	{
		i=10;
	}
}
Output:
Compile successfully.
```

If we are performing initialization anywhere else we will get compile time error.

Example:

```
class Test
{
	final static int i;
	public static void main(String args[])
	{
		i=10;
	}
}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:5: cannot assign a value to final variable i
i=10;
```

Final local variables:

- To meet temporary requirement of the Programmer, sometimes we can declare the variable inside a method or block or constructor such type of variables are called local variables.
- For the local variables, jvm won't provide any default value. Compulsory, we should perform initialization explicitly before using that variable.

Example:

```
class Test
{
	public static void main(String args[])
	{
		int i;
		System.out.println("hello");
	}
}
Output:
D:\Java>javac Test.java
D:\Java>java Test
Hello
```

Example:

```
class Test
{
	public static void main(String args[])
	{
		int i;
		System.out.println(i);
	}
}
Output:
Compile time error.
```

```
D:\Java>javac Test.java
Test.java:5: variable i might not have been initialized
System.out.println(i);
```

Even though local variable declared as the final, before using only, we should perform initialization.

Example:

```
class Test
{
        public static void main(String args[])
        {
                final int i;
                System.out.println("hello");
        }
}
Output:
D:\Java>javac Test.java
D:\Java>java Test
hello
```

Note: The only applicable modifier for local variables is final. If we are using any other modifier we will get compile time error.

Example:

```
class Test
{
public static void main(String args[])
{
private int x=10;————(invalid)
public int x=10; ————(invalid)
volatile int x=10;————(invalid)
transient int x=10;—— (invalid)
final int x=10;————(valid)
}
}
```

```
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:5: illegal start of expression
private int x=10;
```

Important points:

1. A final variable means you can't change its value.
2. A final method means you can't override the method.
3. A final class means you can't extend the class (i.e. you can't make a subclass).

Abstract Modifier

Agenda:

Abstract Modifier
- Abstract Methods
- Abstract class

Abstract modifier:

Abstract is the modifier applicable only for methods and classes but not for variables.

a) Abstract methods:

- Even though we don't have implementation still we can declare a method with abstract modifier. i.e., abstract methods have **only declaration** but **not implementation**.
- Hence abstract method declaration should compulsory ends with **semicolon**.

Example:

```
public abstract void methodOne();  ------> valid
public abstract void methodOne(){} ------> invalid
```

Child classes are responsible to provide implementation for parent class abstract methods.

Program:

vehicle.java

```
abstract class vehicle
{
public abstract int getNoOfWheels();
}
```

Bus.java

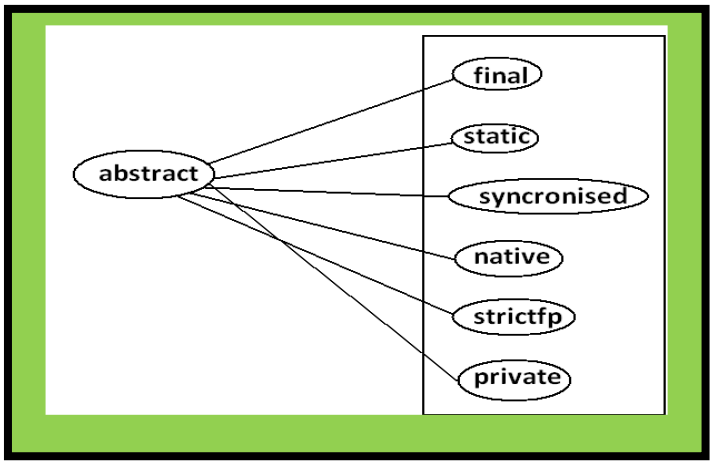
```
class Bus extends vehicle
{
public int getNoOfWheels(){
return 7;
}
}
```

Auto.java

```
class Auto extends vehicle
{
public int getNoOfWheels(){
return 3;
}
}
```

- The main advantage of abstract methods is , by declaring abstract method in parent class we can provide guide lines to the child class such that which methods they should compulsory implement.
- Abstract method never talks about implementation.
- If any modifier talks about implementation then the modifier will be enemy to abstract and that is always illegal combination for methods.

The following are the various illegal combinations for methods.
Diagram:



All the 6 combinations are illegal.

b) Abstract class:

For any java class, if we are not allowed to create an object, such type of class we have to declare with abstract modifier i.e., for abstract class, instantiation is not possible.
Example:

```
abstract class Test
{
	public static void main(String args[])
	{
		Test t=new Test();
	}
}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:4: Test is abstract; cannot be instantiated
Test t=new Test();
```

What is the difference between abstract class and abstract method ?

- If a class contain at least one abstract method, then compulsory the corresponding class should be declared with abstract modifier. Because implementation is not complete and hence we can't create object of that class.
- Even though class doesn't contain any abstract methods, still we can declare the class as abstract i.e., an abstract class can contain zero no of abstract methods also.

Example1: HttpServlet class is abstract but it doesn't contain any abstract method.
Example2: Every adapter class is abstract but it doesn't contain any abstract method.
Example1:

```
class Parent
{
	public void methodOne();
}
Output:
Compile time error.
D:\Java>javac Parent.java
Parent.java:3: missing method body, or declare abstract
public void methodOne();
```

Example 2:

```
class Parent
{
	public abstract void methodOne(){}
}
Output:
Compile time error.
Parent.java:3: abstract methods cannot have a body
public abstract void methodOne(){}
```

Example3:

```
class Parent
{
	public abstract void methodOne();
}
Output:
Compile time error.
D:\Java>javac Parent.java
Parent.java:1: Parent is not abstract and does not  override abstract
method methodOne() in Parent
class Parent
```

If a class extends any abstract class, then compulsory we should provide implementation for every abstract method of the parent class otherwise we have to declare child class as abstract.

Example:

```
abstract class Parent
{
	public abstract void methodOne();
```

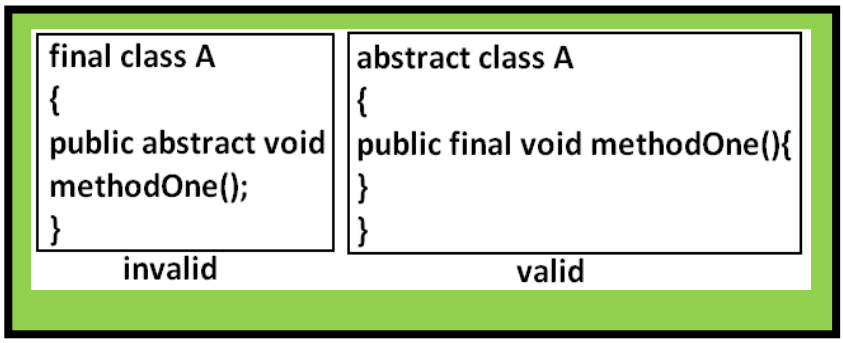
```
        public abstract void methodTwo();
}
class child extends Parent
{
        public void methodOne(){}
}
Output:
Compile time error.
D:\Java>javac Parent.java
Parent.java:6: child is not abstract and does not override abstract
method methodTwo() in Parent
class child extends Parent
```

If we declare class child as abstract then the code compiles fine but child of child is responsible to provide implementation for methodTwo( ).

What is the difference between final and abstract ?

- For abstract methods, compulsory we should override in the child class to provide implementation. Whereas for final methods, we can't override. Hence abstract final combination is illegal for methods.
- For abstract classes, we should compulsory create child class to provide implementation whereas for final class we can't create child class. Hence final abstract combination is illegal for classes.
- Final class cannot contain abstract methods whereas abstract class can contain final method.

Example:



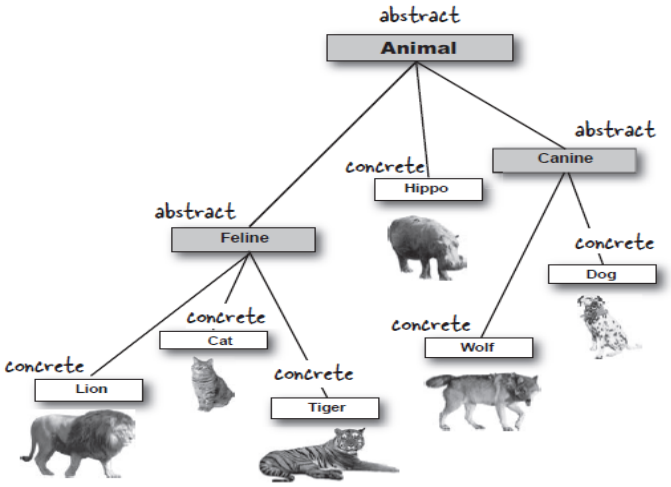
Note:

Usage of abstract methods, abstract classes and interfaces is always good Programming practice.

Abstract class vs Concrete class:

The only real difference is that a concrete class can be instantiated because it provides (or inherits) the implementation for all of its methods. An abstract class cannot be instantiated because at least one method has not been implemented.

A class that's not abstract is called a ***concrete*** class. In the Animal inheritance tree, if we make Animal, Canine, and Feline abstract, that leaves Hippo, Wolf, Dog, Tiger, Lion, and Cat as the concrete subclasses.



Important points:

1. An abstract method has no body! `public abstract void eat();`
2. If you declare an abstract method, you MUST mark the class abstract as well. You cannot have an abstract method in a non-abstract class.
3. You MUST implement all abstract methods.
4. Implementing an abstract method is just like overriding a method.

Interfaces

Agenda

Interfaces
- Interface declarations and implementations
- Extends vs implements
- Interface methods
- Interface variables
- Interface vs abstract class vs concrete class
- Difference between interface and abstract class?

Interfaces:

- Interface is also one of the type of class it contains only abstract methods and Interfaces are not alternative for abstract class, it is extension for abstract classes.
- For the interfaces, the compiler will generates .class files.
- Interfaces giving the information about the functionalities and these are not giving the information about internal implementation.
- Inside the source file, it is possible to declare any number of interfaces. And we are declaring the interfaces by using **interface** keyword.

  Syntax:-interface interface-name { }

  Eg: interface it1 { }

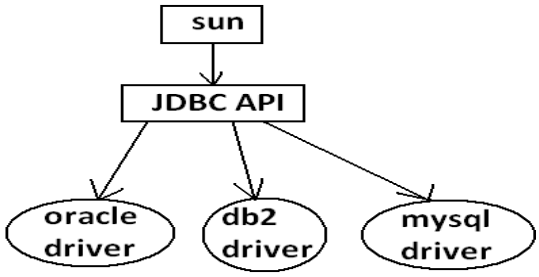
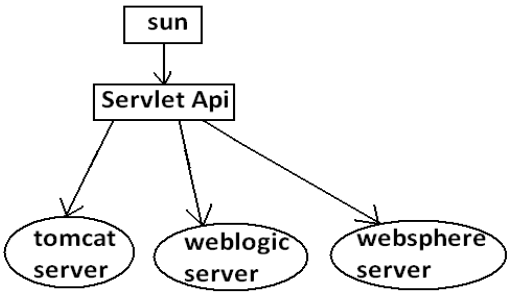
BOTH EXAMPLES ARE SAME

```
interface it1
{
        void m1( );
        void m2( );
        void m3( );
}
```

```
abstract interface it1
{
        public abstract void m1( );
        public abstract void m2( );
        public abstract void m3( );
}
```

Note: - If we are declaring or not each and every interface method by default public abstract. And the interfaces are by default abstract hence for the interfaces object creation is not possible.

Def 1 : Any Service Requirement Specification (SRS) is called an interface.

Example1: Sun people responsible to define JDBC API and database vendor will provide implementation for that. Diagram:  	**Example2:** Sun people define Servlet API to develop web applications web server vendor is responsible to provide implementation. Diagram: 

Def 2: From the client point of view an interface define the set of services what is expecting. From the service provider point of view an interface defines the set of services what is offering. Hence an interface is considered as a contract between client and service provider.
Example: ATM GUI screen describes the set of services what bank people offering, at the same time the same GUI screen the set of services what customer is expecting hence this GUI screen acts as a contract between bank and customer.

Def 3: Inside interface every method is always abstract whether we are declaring or not hence interface is considered as 100% pure abstract class.

Summary def: Any service requirement specification (SRS) or any contract between client and service provider or 100% pure abstract classes is considered as an interface.

Declaration and implementation of an interface:

Note1:

Whenever we are implementing an interface, compulsory for every method of that interface we should provide implementation otherwise we have to declare class as abstract in that case child class is responsible to provide implementation for remaining methods.

Note2:

Whenever we are implementing an interface method, compulsory it should be declared as public otherwise we will get compile time error.

Extends vs implements:

A class can extends only one class at a time.

Example:

```
class One
{
	public void methodOne()
	{
	}
}
class Two extends One
{
}
```

A class can implements any no. of interfaces at a time.

Example:

```
interface One
{
	public void methodOne();
}
interface Two
{
	public void methodTwo();
}
class Three implements One,Two
{
	public void methodOne()
	{
	}
	public void methodTwo()
	{
	}
}
```

A class can extend a class and can implement any no. of interfaces simultaneously.

```
interface One
{
	void methodOne();
}
class Two
{
	public void methodTwo()
	{
	}
}
class Three extends Two implements One
```

```
{
        public void methodOne()
        {
        }
}
```

An interface can extend any no. of interfaces at a time.

Example:

```
interface One
{
        void methodOne();
}
interface Two
{
        void methodTwo();
}
interface Three extends One,Two
{
}
```

Which of the following is true?

1. A class can extend any no.of classes at a time.(FALSE)
2. An interface can extend only one interface at a time.(FALSE)
3. A class can implement only one interface at a time.(FALSE)
4. A class can extend a class and can implement an interface but not both simultaneously.(FALSE)
5. An interface can implement any no.of interfaces at a time.(FALSE)
6. None of the above.

Ans: 6

Consider the expression X extends Y for which of the possibility of X and Y this expression is true?

1. Both x and y should be classes.
2. Both x and y should be interfaces.
3. Both x and y can be classes or can be interfaces.
4. No restriction.

Ans: 3

X extends Y, Z ?
X, Y, Z should be interfaces.

X extends Y implements Z ?
X, Y should be classes.
Z should be interface.

X implements Y, Z ?
X should be class.
Y, Z should be interfaces.

X implements Y extend Z ?

Example:

```
interface One
{
}
class Two
{
}
class Three implements One extends Two
{
}
Output:
Compile time error.
D:\Java>javac Three.java
Three.java:5: '{' expected
class Three implements One extends Two{
```

Interface methods:

- Every method present inside interface is always public and abstract whether we are declaring or not. Hence inside interface the following method declarations are equal.

```
void methodOne();
public Void methodOne();
abstract Void methodOne();            Equal
public abstract Void methodOne();
```

> **public:** To make this method available for every implementation class.
> **abstract:** Implementation class is responsible to provide implementation .

As every interface method is always public and abstract we can't use the following modifiers for interface methods.
Private, protected, final, static, synchronized, native, strictfp.

Inside interface which method declarations are valid?

1. public void methodOne( ){} (NOT VALID)
2. private void methodOne( ); (NOT VALID)
3. public final void methodOne( ); (NOT VALID)
4. public static void methodOne( ); (NOT VALID)
5. public abstract void methodOne( ); (VALID)

Ans: 5

Interface variables:

- An interface can contain variables.
- The main purpose of interface variables is to define **requirement level constants**.
- Every interface variable is always **public static and final** whether we are declaring or not.

Example:

```
interface interf
{
        int x=10;
}
```

public: To make it available for every implementation class.
static: Without existing object also we have to access this variable.
final: Implementation class can access this value but cannot modify.

Hence inside interface the following declarations are equal.

```
int x=10;
public int x=10;
static int x=10;
final int x=10;                                  Equal
public static int x=10;
public final int x=10;
static final int x=10;
public static final int x=10;
```

- As every interface variable by default **public static final** we can't declare with the following modifiers.
  - Private
  - Protected
  - Transient
  - Volatile

- For the interface variables, compulsory we should perform initialization at the time of declaration only otherwise we will get compile time error.

Example:

```
interface Interf
{
int x;
}
Output:
Compile time error.
D:\Java>javac Interf.java
Interf.java:3: = expected
int x;
```

Which of the following declarations are valid inside interface ?

1. int x; (NOT VALID)
2. private int x=10; (NOT VALID)
3. public volatile int x=10; (NOT VALID)
4. public transient int x=10; (NOT VALID)
5. public static final int x=10; (VALID)

Ans: 5

Interface variables can be accessed from implementation class but cannot be modified.

Example:

```
interface Interf
{
        int x=10;
}
```

Example 1:

```
class Test implements Interf
{
public static void main(String args[]){
x=20;
System.out.println("value of x"+x);
}
}
output:
compile time error.
D:\Java>javac Test.java
Test.java:4: cannot assign a value to final variable x
x=20;
```

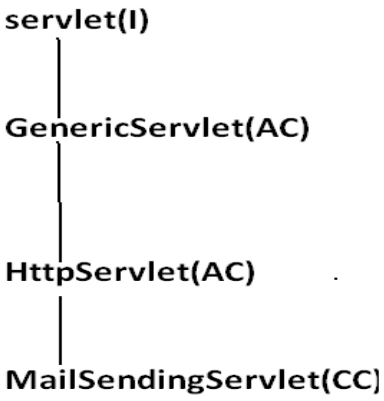
Example 2:

```
class Test implements Interf
{
	public static void main(String args[])
	{
		int x=20;
		//here we declaring the variable x.
		System.out.println(x);
	}
}
Output:
D:\Java>javac Test.java
D:\Java>java Test
20
```

What is the difference between interface, abstract class and concrete class? When we should go for interface, abstract class and concrete class?

- If we don't know anything about implementation just we have requirement specification then we should go for interface(I).
- If we are talking about implementation but not completely (partial implementation) then we should go for abstract class(AC).
- If we are talking about implementation completely and ready to provide service then we should go for concrete class(CC).

Example:



What is the Difference between interface and abstract class ?

interface	Abstract class
If we don't' know anything about implementation just we have requirement specification then we should go for interface.	If we are talking about implementation but not completely (partial implementation) then we should go for abstract class.
Every method present inside interface is always public and abstract whether we are declaring or not.	Every method present inside abstract class need not be public and abstract.
We can't declare interface methods with the modifiers private, protected, final, static, synchronized, native, strictfp.	There are no restrictions on abstract class method modifiers.
Every interface variable is always public static final whether we are declaring or not.	Every abstract class variable need not be public static final.
Every interface variable is always public static final we can't declare with the following modifiers. Private, protected, transient, volatile.	There are no restrictions on abstract class variable modifiers.
For the interface variables compulsory we should perform initialization at the time of declaration otherwise we will get compile time error.	It is not require to perform initialization for abstract class variables at the time of declaration.
Inside interface we can't take static and instance blocks.	Inside abstract class we can take both static and instance blocks.
Inside interface we can't take constructor.	Inside abstract class we can take constructor.

We can't create object for abstract class but abstract class can contain constructor what is the need ?

Abstract class constructor will be executed when ever we are creating child class object to perform initialization of child object.

Example:

```
class Parent
{
	Parent()
	{
		System.out.println(this.hashCode());
	}
}
class child extends Parent
{
	child()
	{
		System.out.println(this.hashCode());
	}
}
```

```
class Test
{
        public static void main(String args[])
        {
                child c=new child();
                System.out.println(c.hashCode());
        }
}
```

Note : We can't create object for abstract class either directly or indirectly.

Every method present inside interface is abstract but in abstract class also we can take only abstract methods then what is the need of interface concept ?

We can replace interface concept with abstract class. But it is not a good programming practice. We are misusing the roll of abstract class. It may create performance problems also.
(this is just like recruiting IAS officer for sweeping purpose)

Example :

```
interface X {
 -------
 -------
}
class Test  implements  X {
  ----------
  ----------
}

Test t=new Test();
        // takes 2 sec
1) performance is high
2) while implementing  X
     we can extend  some
     other classes
```

```
abstract class X  {
   -----------------
   -----------------
}
class Test extends  X {
  ------------------
  ------------------
}

Test t=new Test();
          //takes  20sec
1) performance is low
2) while extending  X we can't
    extend  any  other classes
```

If every thing is abstract, then it is recommended to go for interface.

Why abstract class can contain constructor where as interface doesn't contain constructor ?

- The main purpose of constructor is to perform initialization of an object i.e., provide values for the instance variables, Inside interface every variable is always static and there is no chance of existing instance variables. Hence constructor is not required for interface.
- But abstract class can contains instance variable which are required for the child object to perform initialization for those instance variables constructor is required in the case of abstract class.