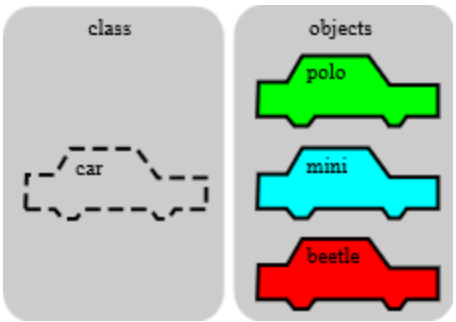
What Is a Class?

A *class* is the blueprint from which individual objects are created.
In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles. A *class* is the blueprint from which individual objects are created.
The following Bicycle class is one possible implementation of a bicycle:

```
class Bicycle {

int cadence = 0;
int speed = 0;
int gear = 1;

voidchangeCadence(int newValue) {
cadence = newValue;
   }

voidchangeGear(int newValue) {
gear = newValue;
   }

voidspeedUp(int increment) {
speed = speed + increment;
   }

voidapplyBrakes(int decrement) {
speed = speed - decrement;
   }

voidprintStates() {
System.out.println("cadence:" +
cadence + " speed:" +
speed + " gear:" + gear);
   }
}
```

The fields cadence, speed, and gear represent the object's state, and the methods (changeCadence, changeGear, speedUp etc.) define its interaction with the outside world.

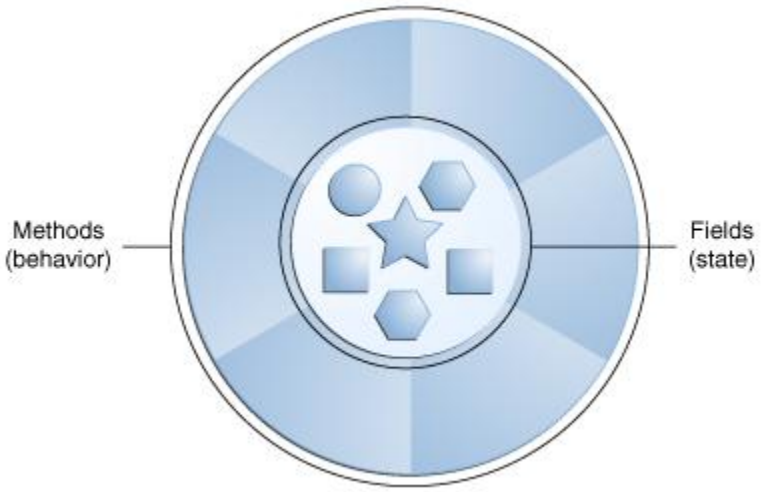


What Is an Object?

Objects are key to understanding *object-oriented* technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.
Real-world objects share three characteristics: They all have *state*, *behavior and identity*. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail).

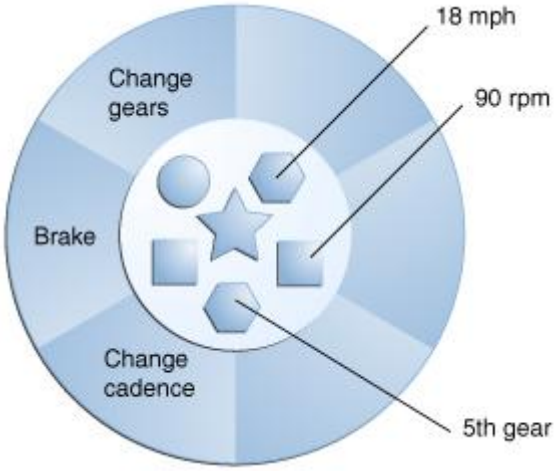
Example2:
Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behavior can this object perform?". Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.



A software object.

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in *fields* (variables in some programming languages) and exposes its behavior through *methods* (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication.



A bicycle modeled as a software object.

Object is the physical existence/construction. Technically object we can call as instance of a class creating object is called instantiation. Instance or non-static variables get memories at that time of object created.

There are four key elements when designing or writing a program using OOP:
Object
Class
Method
Attribute

-In java Object having 3 characteristics those are

1) State
2) Behavior
3) Identity

1) State
- non static or instance variable values technically calling as state of object.
-if non static or instance variables values change that object state also changed.
-objects are created in heap area of jvm memory.

2) Behaviors
-methods of class are technically call it as Behavior.
-object behaviors depend on the methods

3) Identity
-For every object JVM creates unique 32 bit of hash code.
-JVM using hashing algorithm for creating hash code for every object.
-hashCode() method returns the hash value of an object, hashCode() is present in java.lang.Object class

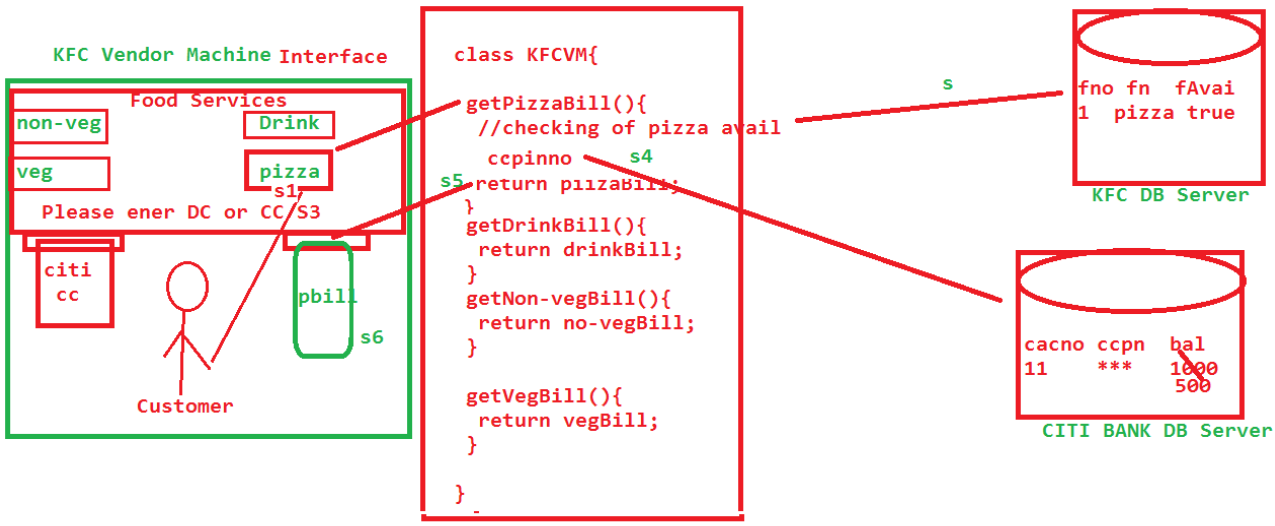
What is data or information hiding?

Data hiding is the principle protecting class data from outside world. In other way data hiding is the prevent direct accessing of class data outside world or class. We can implement data hiding in a class by declaring the data member of a clas s as private.

Example:
```
classAcc
ount{
private double accountBal;
private int accountPassword;
}
```

Abstraction:

Hide internal implementation and just highlight the set of services, is called abstraction. By using abstract classes and interfaces we can implement abstraction.



By using KFC GUI screen exposing KFC services to customers without exposing internal implementation.

Advantages of Abstraction:

1) We can achieve security as we are not highlighting our internal implementation.

2)Enhancement will become very easy because without effecting end user we can able to perform any type of changes in our internal system.

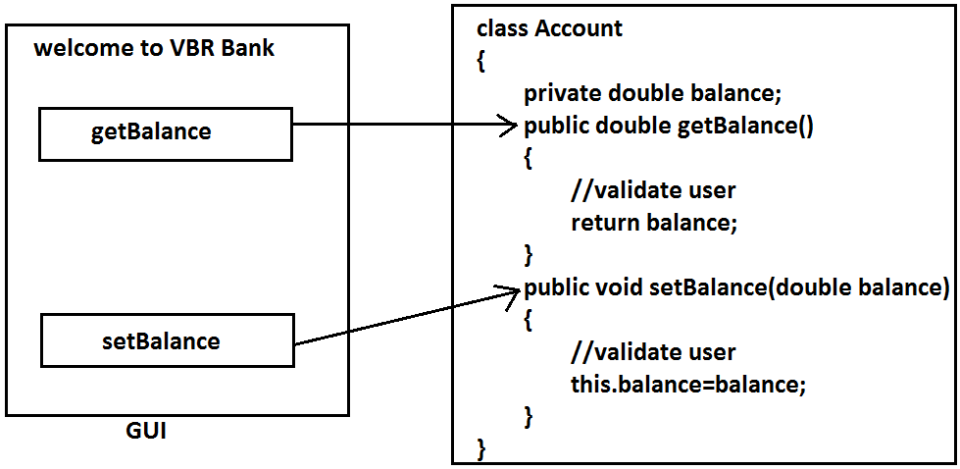
3) It provides more flexibility to the end user to use system very easily.

4) It improves maintainability of the application.

Encapsulation

It is the process of Encapsulating data and corresponding methods into a single module or unit. If any java class follows data hiding and abstraction such type of class is said to be encapsulatedclass.

Encapsulation=Datahiding+Abstraction



Inheritance (IS-A relationship)

Inheritance is a mechanism in which one class acquires the property of another class. For example, a child inherits the characterstics of his/her parents. With inheritance, we can reuse the fields and methods of the existing class. Hence, inheritance facilitates Reusability and is an important concept of OOPs. A child inherits visible properties and methods from its parent while adding additional properties and methods of its own.

Inheritance also called as is-a relationship, the main advantage of IS-A relationship is reusability.

Subclasses and superclasses can be understood in terms of the is a relationship. A subclass is a more specific instance of a superclass. For example, an orange is asweet fruit, which is a fruit. A shepherd is a dog, which is an animal. A villain is a woodwind instrument, which is a musical instrument. If the is a relationship does not exist between a subclass and superclass, you should not use inheritance. An orange is a fruit; so it is okay to write an Orange class that is a subclass of a Fruit class. As a contrast, a kitchen has a sink. It would not make sense to say a kitchen is a sink or that a sink is a kitchen.

Q) How inheritance can be implement in java

Inheritance can implemented in java by using two keywords**extends** and **implements**.

1) **extends**

extends can implemented between two classes and two interfaces

class A{

```
}

class B extends A{
}

interface i{
}
interface j extends i{
}
```

2) implements

implementscan implemented between one classes and one interfaces

```
interface i{
}
class A implements i {
}
```

Look at the ChocolateCake class below:

```
packagecom.jh.kumar.ineritance;

publicclass ChocolateCake {

	public ChocolateCake() {
		System.out.println("Constructor of ChocolateCake");
	}

	publicvoid bake() {
		System.out.println("The chocolate cake is baking");
	}

	publicvoid frost() {
		System.out.println("The chocolate cake now has frosting");
	}

	publicvoidputCandlesOnCake(intnumberOfCandles) {
		System.out.println("Putting " + numberOfCandles + " candles on      the cake. ");
	}
}
```

It's a simple class with three methods: bake(), frost(), and putCandlesOnCake(). You would have a good chocolate cake that you could put candles on after instantiating this class. As fun as it might be, do you really want or need candles on every chocolate cake you make? Probably not. Typically, candles only go on birthday cakes. Classes should only include the properties and methods that they need. Not every instance of ChocolateCake needs candles, so the class shouldn't include that method. Instead, you should write a separate BirthdayCake class that can have candles. To keep this first example simple, BirthdayCake will inherit from ChocolateCake.

Look at a revised version of ChocolateCake:

```
packagecom.jh.kumar.ineritance;

publicclass ChocolateCake {

	public ChocolateCake() {
		System.out.println("Constructor of ChocolateCake");
	}

	publicvoid bake() {
		System.out.println("The chocolate cake is baking");
	}

	publicvoid frost() {
		System.out.println("The chocolate cake now has frosting");
	}

}
```

The putCandlesOnCake() method has been removed from ChocolateCake. Look at BirthdayCake below. It inherits the properties and methods of ChocolateCake and adds the ability to put candles on the cake.

```
packagecom.jh.kumar.ineritance;

publicclassBirthDayCakeextendsChocolateCake {

	publicBirthDayCake() {
		System.out.println("Constructor of BirthdayCake");

	}

	publicvoidputCandlesOnCake(intnumberOfCandles) {
		System.out.println("Putting " + numberOfCandles +
				" candles on the birthday cake.");
	}
	publicstaticvoid main(String[] args) {
		BirthDayCakebirthDayCake = newBirthDayCake();
		birthDayCake.bake();
		birthDayCake.frost();
		birthDayCake.putCandlesOnCake(10);
	}

}
```

Output:

```
Constructor of ChocolateCake
Constructor of BirthdayCake
The chocolate cake is baking
The chocolate cake now has frosting
Putting 10 candles on the birthday cake.
```

Notice that BirthdayCake does not explicitly include a method named bake(), but you can still call that method on the instance of BirthdayCake because a subclass inherits all the public methods of its superclass. Since bake() is a public method in ChocolateCake, bake() is also a method in BirthdayCake. And the result of the method—The chocolate cake is baking—is the same regardless of the instance on which you call the method.

Multilevel Inheritance:

```
packagecom.jh.kumar.ineritance;

class ChocolateCake {

	public ChocolateCake() {
		System.out.println("Constructor of ChocolateCake");
	}

	publicvoid bake() {
		System.out.println("The chocolate cake is baking");
	}

	publicvoid frost() {
		System.out.println("The chocolate cake now has frosting");
	}

}

classBirthDayCakeextendsChocolateCake {

	publicBirthDayCake() {
		System.out.println("Constructor of BirthdayCake");

	}

	publicvoidputCandlesOnCake(intnumberOfCandles) {
		System.out.println("Putting " + numberOfCandles +
				" candles on the birthday cake.");
	}
}
```

```
publicclassMarriageCakeextendsBirthDayCake{

	publicvoidputCouplesImgOnCake(){
		System.out.println("couples image arranged on cake..");
	}
	publicstaticvoid main(String[] args) {
		MarriageCakemarriageCake = newMarriageCake();
```

```
        marriageCake.bake();
        marriageCake.frost();
        marriageCake.putCandlesOnCake(10);
        marriageCake.putCouplesImgOnCake();
    }

}
```

Output:

```
Constructor of ChocolateCake
Constructor of BirthdayCake
The chocolate cake is baking
The chocolate cake now has frosting
Putting 10 candles on the birthday cake.
couples image arranged on cake..
```

Q) How JVM can load super class when sub class is loaded?

By using **extends** keyword.

Q) Need of super() ?

super() method is used to call the super class constructor.

Q) Need of super keyword ?

The **super** keyword used to access the members of the super class (super.a)

Q) Difference between this and super keywords?

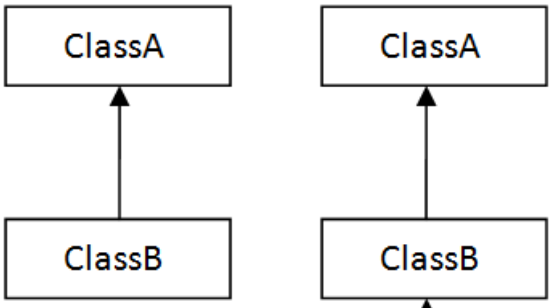
this	**super**
1) it is non static reference variable	1) it is non static reference variable
2) this keyword hold current object references	2) super keyword points to super class variables
3) it is separate the method arg name and variables	3) it is separate super class variable and sub class level class variable
4) this keyword used only in non static method	4)super keyword used only in non static method

Types of Inheritances:

On the basis of class, there can be three types of inheritance in java

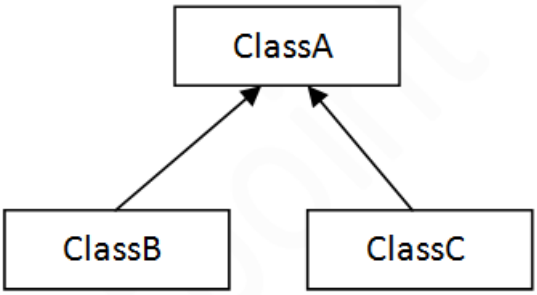
1. **Single inheritance**
2. **Multilevel inheritance**
3. **Hierarchical**

Multiple and hybrid inheritance is supported through interfaces only.

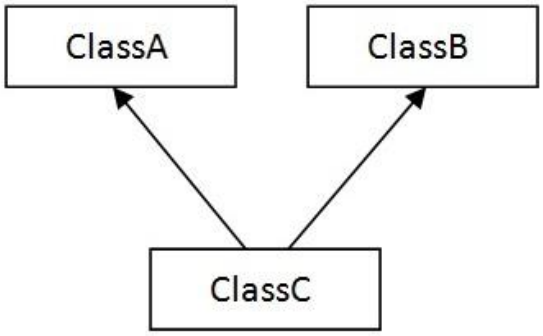


1) Single

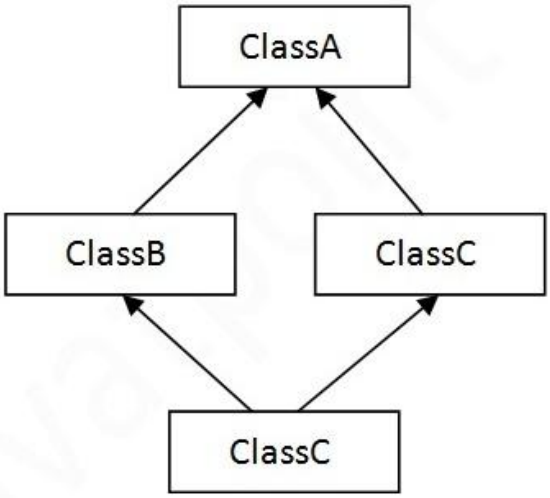
2) Multilevel



3) Hierarchical



4) Multiple



5) Hybrid

.

Single Inheritance

```
class Super
{
	void m1()
	{
		System.out.println("sup class");
	}
}
class Single extends Super
```

```
{
	void m2()
	{
		System.out.println("sub class");
	}
	public static void main(String args[]){
			System.out.println("main");
			Single obj=new Single();
			obj.m1();
			obj.m2();
	}
}
```

Multilevel Inheritance

```
class A
{
	void m1()
	{
		System.out.println("A class");
	}
}
class B extends A
{
	void m2()
	{
		System.out.println("B class");
	}
}
classMultilevelI extends B
{
	void m3()
	{
		System.out.println("Multilevel class");
	}
	public static void main(String args[]){
			System.out.println("main");
			MultilevelI obj=new MultilevelI();
			obj.m1();
			obj.m2();
			obj.m3();
	}
}
```

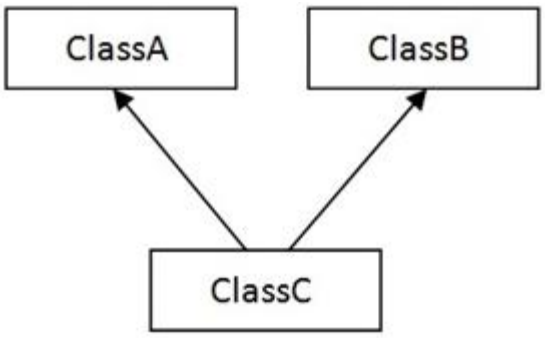
Hierarchical Inheritance

```
class A
{
	void m1()
	{
		System.out.println("A class");
	}
}
```

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

public class HierachicalI extends A
{
	void m3()
	{
		System.out.println("Multilevel class");
	}
	public static void main(String args[]){
			System.out.println("main");
			HierachicalIobj=new HierachicalI();
			obj.m1();
			obj.m3();
	}
}
```

Multiple Inheritances

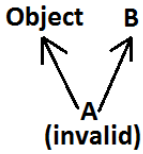
Having more than one Parent class at the same level is called multiple inheritances. In java any class can extends only one class at a time and can't extends more than one class simultaneously hence java won't provide support for multiple inheritance.



4) Multiple

Example

```
class A{}
class B{}
class C extends A,B
{}
```
(invalid)



But an interface can extends any no. Of interfaces at a time hence java provides support for multiple inheritances through interfaces.

Example

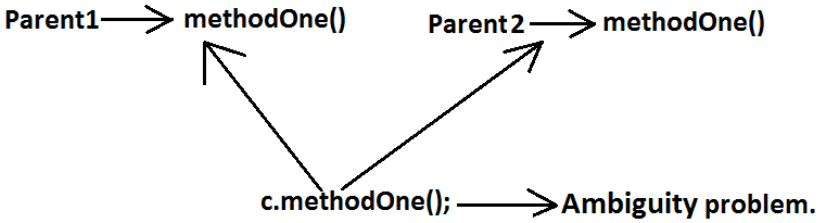
```
interface A{}
interface B{}
interface C extends A,B{}
```

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

Why java won't provide support for multiple inheritance?

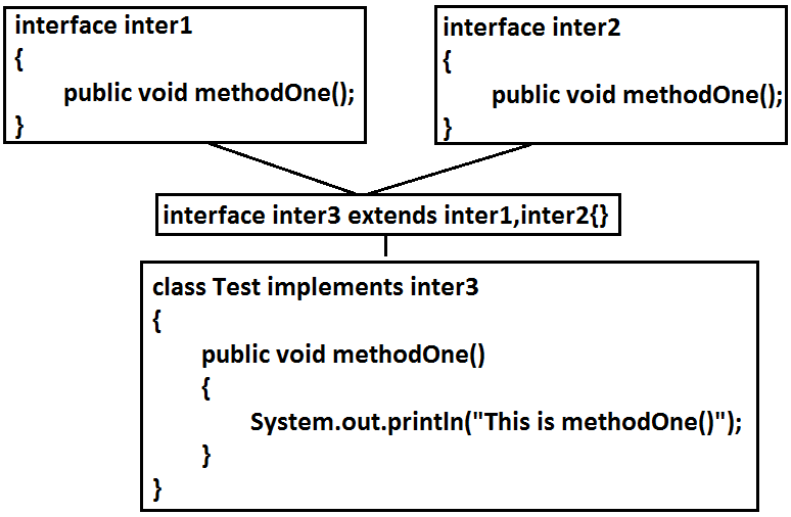
There may be a chance of raising ambiguity problems.

Example:



Why ambiguity problem won't be there in interfaces?

- Interfaces having dummy declarations and they won't have implementations hence no ambiguityproblem.



Example of Multiple inheritance using interface

```
Package com.jh.kumar.ineritance;

interface BreakI {

	publicvoidhitBreak();
}

Interface MediaPlayerI {
	publicvoidplaySong();
}

Public class SwiftCar implements BreakI, MediaPlayerI {

	@Override
	publicvoidplaySong() {
		System.out.println("Play the nice songs..");
	}

	@Override
	Public void hitBreak() {
		System.out.println("Hit the break...");
	}

	Public static void main(String[] args) {
		SwiftCarswiftCar = new SwiftCar();
		swiftCar.playSong();
		swiftCar.hitBreak();
	}
}
```

Output:

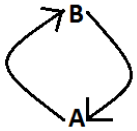
```
Play the nice songs..
Hit the break...
```

Cyclic inheritance:

Cyclic inheritance is not allowed injava.

Example 1:

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



Example 2:

class A extends A{} —C.E→ **cyclic inheritance involving A**

HAS-A relationship:

1) HAS-A relationship is also known as composition (or) aggregation.
2) There is no specific keyword to implement HAS-A relationship but mostly we can use newoperator.
3) The main advantage of HAS-A relationship is reusability.

Example:

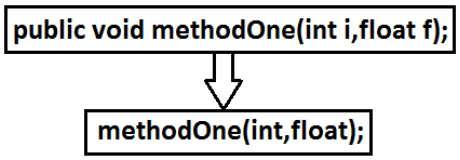
```
class Engine
{
//engine specific functionality
}
class Car
{
Engine e=new Engine();
//.....;
//.....;
//.....;
}
```

Class Car HAS-A engine reference. HAS-A relationship increases dependency between the components and creates maintains problems.

Method signature:

In java method signature consists of name of the method followed by argument types.

Example:



In java return type is not part of the method signature.Compiler will use method signature while resolving method calls. Within the same class we can't take 2 methods with the same signature otherwise we will get compile time error.

Example:

```
public void methodOne(){}
publicintmethodOne()
{
        return 10;
}
```

Output:

```
Compile time error
methodOne() is already defined in Test
```

Overloading:

Two methods are said to be overload if and only if both having the same name but different argument types.

Example:
getEmpId(inteId)
getEmpId(String eId)
getEmpId(long eId)

Having the same name and different argument types is called method overloading. All above methods are considered as overloaded methods. Having overloading concept in java reduces complexity of the programming.

Example:

```
class Employee {
	publicvoidgetEmpName() {
		System.out.println("no-arg method");
	}
	publicvoidgetEmpName(inteId) {
		System.out.println("int-arg method");
	}
	publicvoidgetEmpName(doubled) {
		System.out.println("double-arg method");
	}

	publicstaticvoid main(String[] args) {
		Employee employee = newEmployee();
		employee.getEmpName();// no-arg method
		employee.getEmpName(10);//int-arg method
		employee.getEmpName(10.20);// double-arg method
	}
}
```

In overloading compiler is responsible to perform method resolution (decision) based on the reference type. Hence overloading is also considered as compile time polymorphism (or) static (or)early biding.

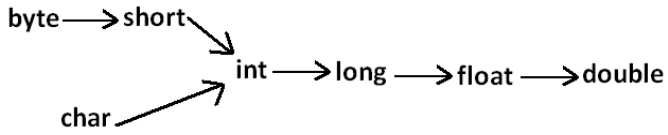
Case 1: Automatic promotion in overloading.

- 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.

Diagram:




```
class Employee {
	publicvoidgetEmpName(inti) {
		System.out.println("int-arg method "+ i);
	}
	// overloaded methods
	publicvoidgetEmpName(floatf) {
		System.out.println("float-arg method "+ f);
	}

	publicstaticvoid main(String[] args) {

		Employee employee = newEmployee();
		employee.getEmpName('a'); //int-arg method
		employee.getEmpName(10l); //float-arg method
		//The method getEmpName(int)in the
		//type Employee is not applicable for the arguments (double)
		//employee.getEmpName(10.5);

	}
}
```

Case 2:

In overloading Child will always get high priority then Parent.

```
class Employee {
	publicvoidgetEmpName(String s) {
		System.out.println("String version");
	}

	// Both methods are said to be overloaded methods.
	publicvoidgetEmpName(Object o) {
		System.out.println("Object version");
	}

	publicstaticvoid main(String[] args) {
		Employee employee = newEmployee();
		employee.getEmpName("pawankalyan");// String version
		employee.getEmpName(new Object());// Object version
		employee.getEmpName(null);// String version
	}
}
```

Output:

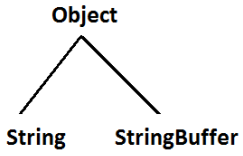
```
String version
Object version
String version
```

Case 3:

```
class Test {
	publicvoidtestMethod(String s) {
		System.out.println("String version  " + s);
	}

	publicvoidmethodOne(StringBuffer  s) {
		System.out.println("StringBuffer version  "+ s);
	}

	publicstaticvoid main(String[] args) {
		Test test = newTest();
		test.testMethod("mahesh");// String version
		test.methodOne(new StringBuffer("prabash"));// StringBuffer version
	}
}
```

Output:

```
String version  mahesh
StringBufferversion  prabash
```



Case 4:

```
class Test {
	publicvoidtestMethod(int i, float f) {
		System.out.println("int-float method");
	}
	publicvoidmethodOne(float f, int i) {
		System.out.println("float-int method");
	}
	publicstaticvoid main(String[] args) {
		Test test = newTest();
		test.testMethod(10, 10.5f);// int-float method
		test.methodOne(10.5f, 10);// float-int method
		// C.E:reference to methodOne is ambiguous, both
		// method methodOne(int,float) in Test and method
		// methodOne(float,int) in Test match
		test.testMethod(10, 10);
		// C.E:The method methodOne(float, int) in
		// the type Test is not applicable
		//for the arguments (float, float)
		test.testMethod(10.5f, 10.5f);
	}
}
```

Case 5:

```
class Test {
	publicvoidtestMethod(int i) {
		System.out.println("general method");
	}

	publicvoidtestMethod(int... i) {
		System.out.println("var-arg method");
	}

	publicstaticvoid main(String[] args) {
		Test t = newTest();
		t.testMethod();// var-arg method
		t.testMethod(10,20);//var-arg method
		t.testMethod(10);//general method

	}
}
```

Output

```
var-arg method
var-arg method
general method
```

Note: In general var-arg method will get less priority that is if no other method matched then only var-arg method will get chance for execution it is almost same as default case inside switch.

Case 5:

```
class Animal {
}

class Monkey extends Animal {
}

class Test {
	publicvoidtestMethod(Animal a) {
		System.out.println("Animal version");
	}

	publicvoidtestMethod(Monkey m) {
		System.out.println("Monkey version");
	}
	publicstaticvoid main(String[] args) {
		Test test = newTest();
		Animal a = newAnimal();
		test.testMethod(a);// Animal version
		Monkey m=newMonkey();
		test.testMethod(m);//Monkey version
		Animal a1=newMonkey();
		test.testMethod(a1);//Animal version
	}
}
```

```
Output:
        Animalversion
        Monkey version
        Animal version
```

In overloading method resolution is always based on reference type and runtime object won't play any role inoverloading.

Overriding:

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. The Parent class method which is overridden is called overridden method. The Child class method which is overriding is called overriding method.

Example 1:

```
class Parent{
public void marriage() {// overridden
System.out.println("arranged marriage");          overridden method
}
}
classChild extends Parent {
public void marriage() {// overriding
System.out.println("love marriage");      overriding method
}
}
class Test
{
publicstatic void main(String[] args){
Parent p=new Parent();
p.marriage();//arranged marriage(parent method)
Child c=new Child();
c.marriage();//love marriage
Parent p1=new Child();
p1.marriage();//love marriage
}
}
```

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.
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.

```
public class OverridingTest {
	int rollno = 20;
	@Override
	public String toString() { //s3 s6
		return "Object State is: "+ this.rollno;//child class impl logic
	}
	public static void main(String[] args) { //s0

		OverridingTest test = new OverridingTest(); //s1

		System.out.println(test); //s2 s4 Object State is: 20
		System.out.println(new OverridingTest()); //s5 s7 Object State is: 20

		String name = new String("arjun"); //s8
		System.out.println(name); //s9

	}

}
```

Rules for overriding:

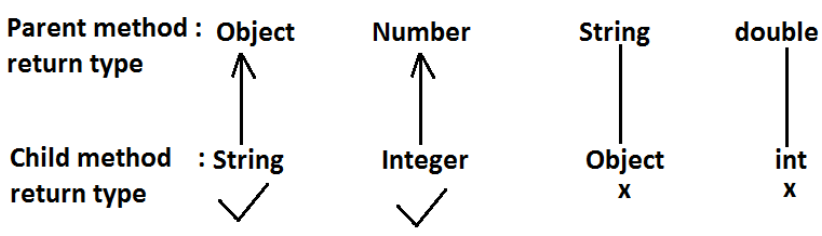
1. In overriding method names and arguments must be same. That is method signature must be same.
2. Until 1.4 version the return types must be same but from 1.5 version onwards co-variant return types are allowed.
3. According to this Child class method return type need not be same as Parent class method return type its Child types also allowed.

Example:

```
class Parent {

	public Object testMethod(){
		return null;
	}
}
public class Test extends Parent {

	@Override
	public String testMethod(){
		return null;
	}

}
```

- It is valid in "1.5" but invalid in"1.4".

Diagram:



Co-variant return type concept is applicable only for object types but not for primitives.

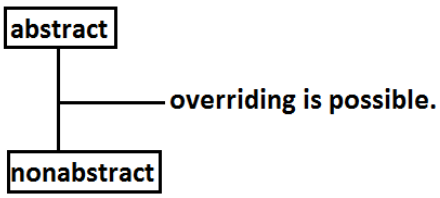
Private methods are not visible in the Child classes hence overriding concept is not applicable for private methods. Based on own requirement we can declare the same Parent class private method in child class also. It is valid but not overriding.

```
class Parent1 {

        private Object testMethod(){
                return null;
        }
}
public class Test extends Parent1 {

        //This is'nt overriding method
        public String testMethod(){
                return null;
        }

}
```

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

We can override Parent class non abstract method as abstract to stop availability of Parent class method implementation to the Child classes.

Diagram:

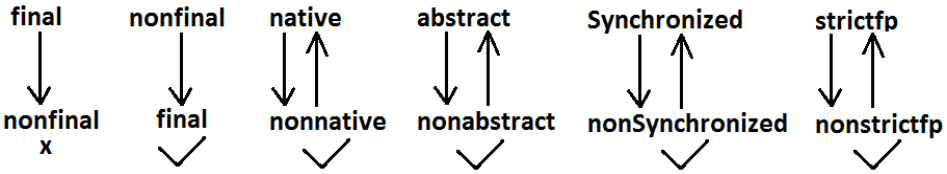


Example:

```
class Parent
{
    public void testMethod()
    {}
}
abstract class Child extends Parent
{
    public abstract void testMethod();
}
```

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

Diagram:



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

Example:

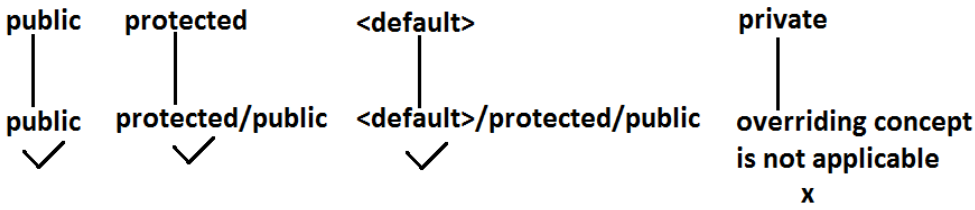
```
class Parent
{
    public void testMethod(){}
}
class Child extends Parent{
    protected void testMethod()
    {}
}
```

Output:

Compile time error

testMethod() in Child cannot override testMethod() in Parent; attempting to assign weaker access privileges; was public

Diagram: ( ) *(private -> default -> protected -> public)*



Checked Vs Un-checked Exceptions:

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

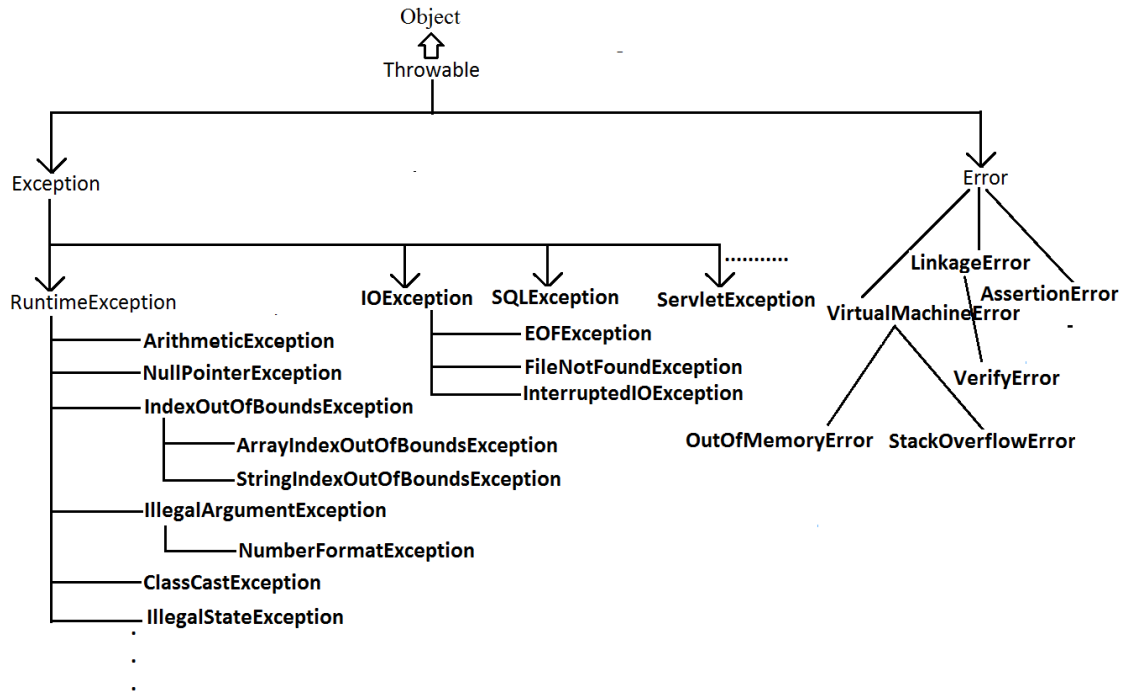


Diagram:

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 timeerror.But there are no restrictions for un-checked exceptions.

Example:

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

Output:

Compile time error

testMethod() 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