Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Exploring Java Basics

Google Cloud Platform

Windows Azure

amazon web services

APACHE kafka
A distributed streaming platform

Apache Software Foundation

APACHE Spark

Java

hadoop

Context And Dependency Injection

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

- Inheritance
- Using super keyword
- The instanceOf Operator
- Method & Constructor overloading
- Method overriding
- @Override annotation
- Using final keyword

SYNERGETICS GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Inheritance

Inheritance in Java

- Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in java by which one class is allow to inherit the features(fields and methods) of another class.

- **Important terminology:**

- **Super Class**
- **Sub Class**
- **Reusability**

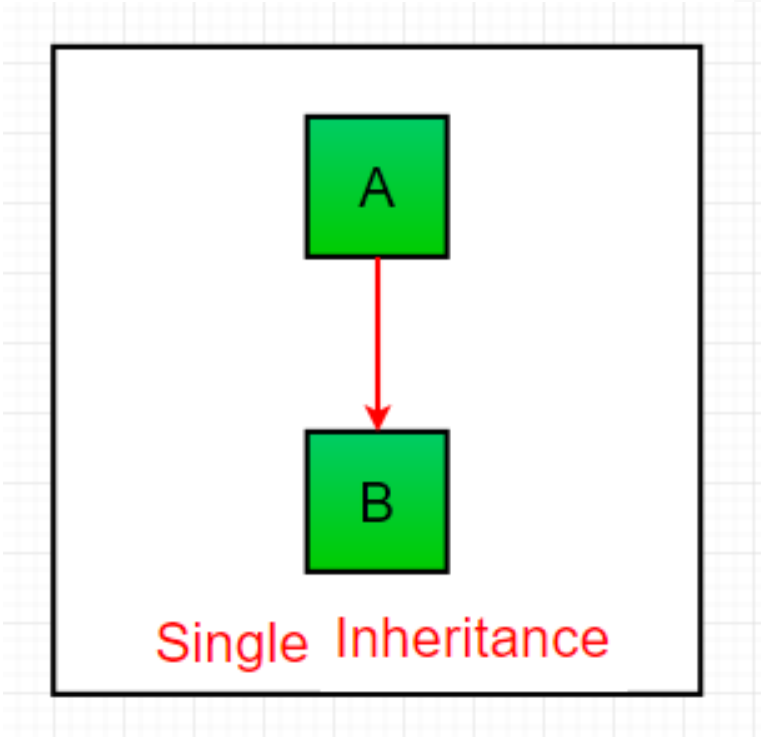
Inheritance in Java

- The keyword used for inheritance is **extends**

```
class derived-class extends base-class
{
   //methods and fields
}
```

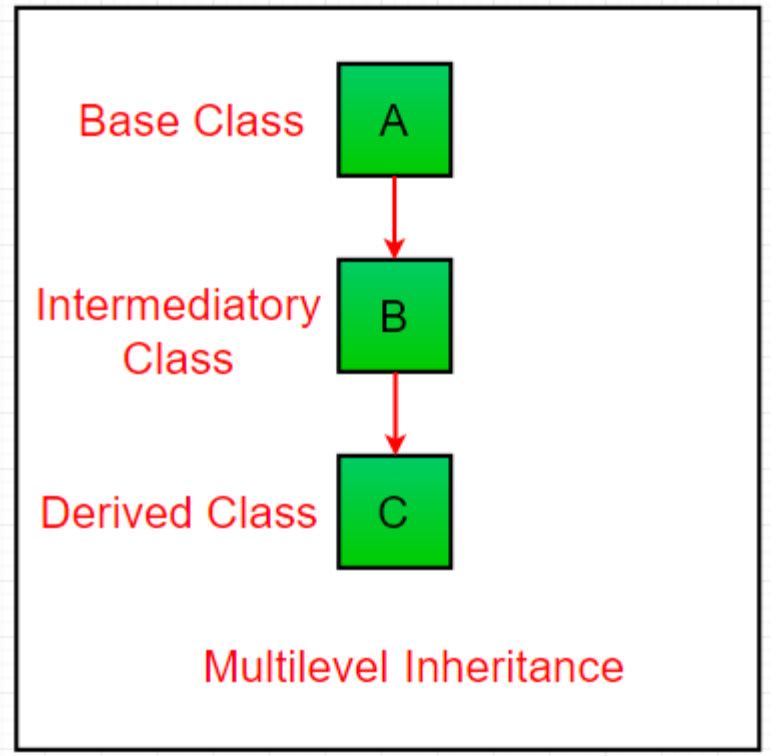
Types of Inheritance in Java

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



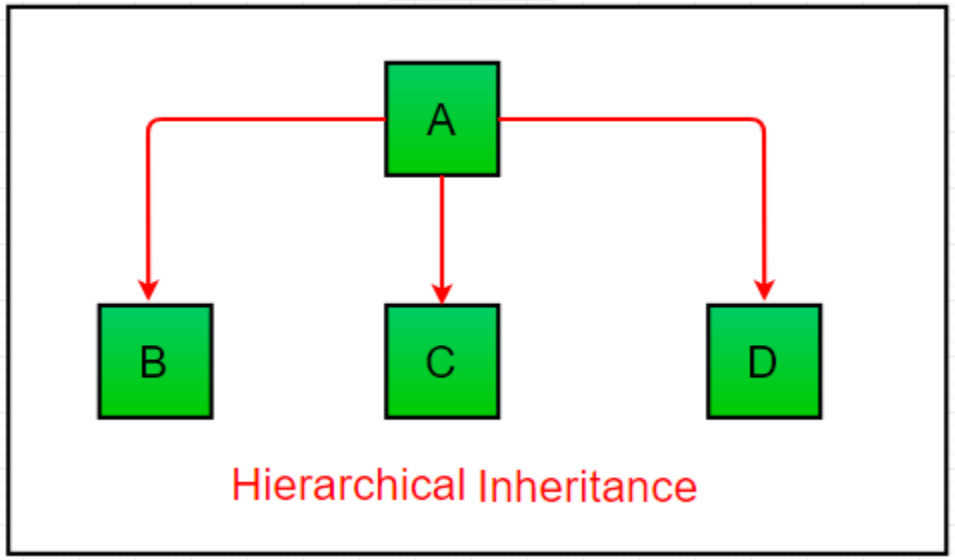
Types of Inheritance in Java

- **Multilevel Inheritance :** In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.



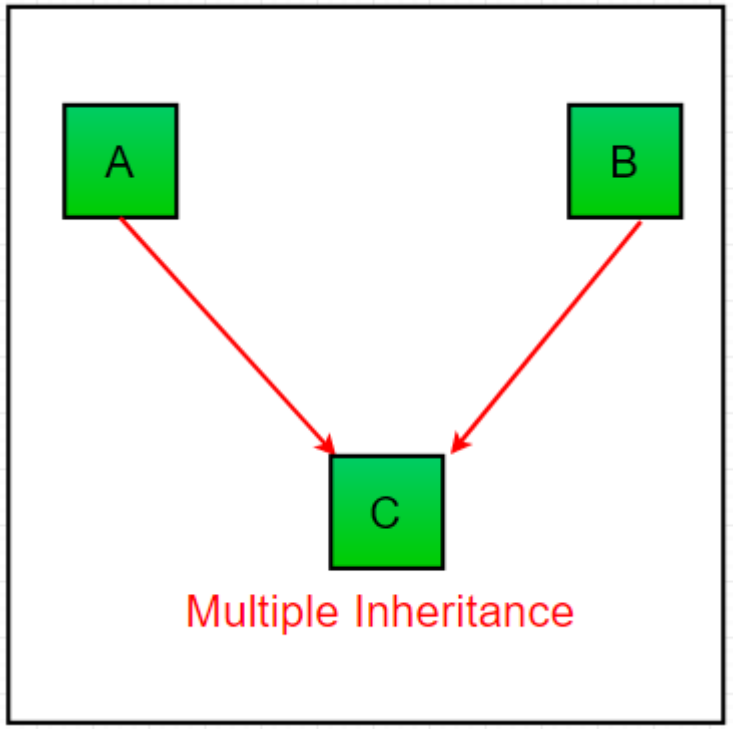
Types of Inheritance in Java

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



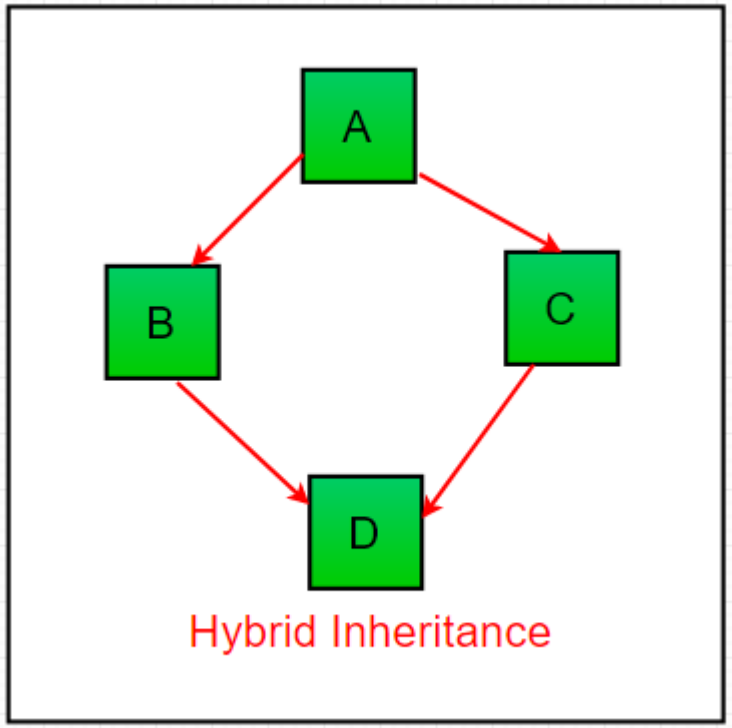
Types of Inheritance in Java

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



Types of Inheritance in Java

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



Important facts about inheritance in Java

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

- **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only one superclass. This is because Java does not support multiple inheritance with classes. Although with interfaces, multiple inheritance is supported by java.

- **Inheriting Constructors:** A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

- **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods(like getters and setters) for accessing its private fields, these can also be used by the subclass.

What all can be done in a Subclass?

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

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

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

The Super Keyword

Super keyword in java

- The **super** keyword in java is a reference variable which is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable

Super keyword in java

Usage of java super Keyword

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

Super keyword in java

- Super can be used to refer immediate parent class instance variable

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

Super keyword in java

- super can be used to invoke parent class method

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
```

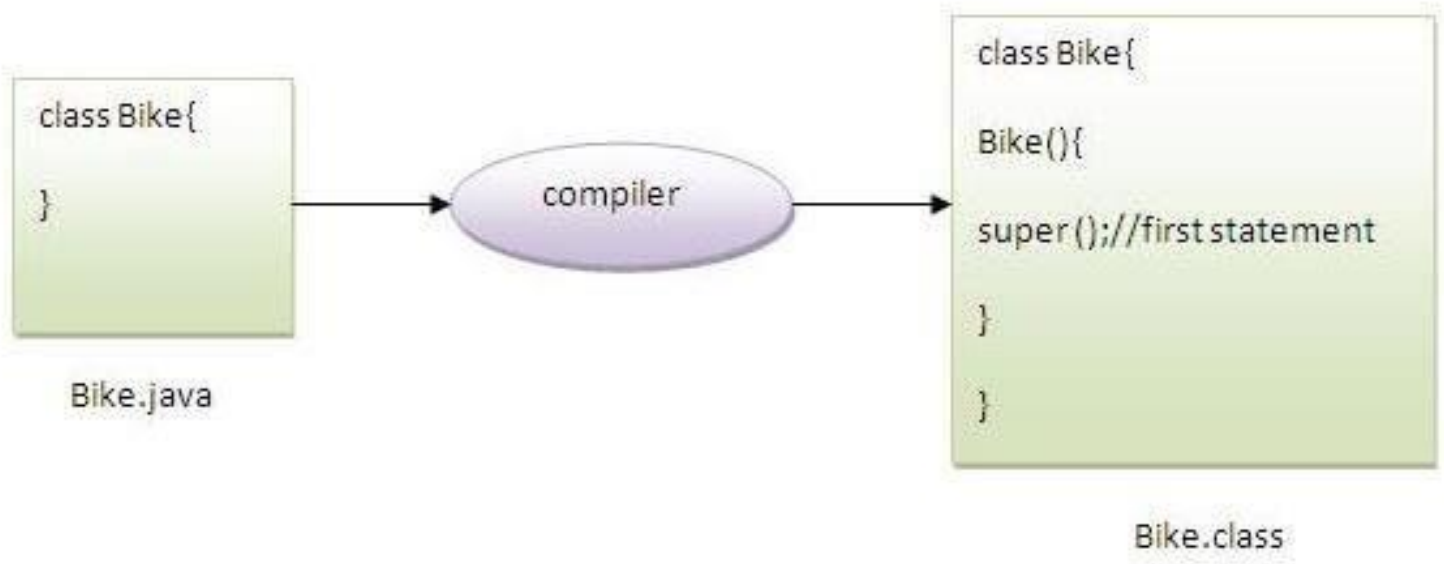
Super keyword in java

super is used to invoke parent class constructor.

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Super keyword in java

- **Note:** super() is added in each class constructor automatically by compiler if there is no super() or this().



Super keyword in java

- Base Class Member initialization

```
class Person{
int id;
String name;
Person(int id,String name){
this.id=id;
this.name=name;
}
}
class Emp extends Person{
float salary;
Emp(int id,String name,float salary){
super(id,name);//reusing parent constructor
this.salary=salary;
}
void display(){System.out.println(id+" "+name+" "+salary);}
}
class TestSuper5{
public static void main(String[] args){
Emp e1=new Emp(1,"ankit",45000f);
e1.display();
}}
```

The *instanceof* Operator

The instanceof Operator

- The java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface).

- The instanceof in java is also known as type comparison operator because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false

```
class Simple1{
 public static void main(String args[]){
 Simple1 s=new Simple1();
 System.out.println(s instanceof Simple1);
 }
}
```

Output: True

```
class Animal{}
class Dog1 extends Animal{//Dog inherits Animal

 public static void main(String args[]){
 Dog1 d=new Dog1();
 System.out.println(d instanceof Animal);//true
 }
}
```

Output True

```
class Animal{}
class Dog1 extends Animal{
 public static void main(String args[]){
 Dog1 d=new Dog1();
 System.out.println(d instanceof Animal);
 }
}
```

Output: True

```
class Dog2{
 public static void main(String args[]){
  Dog2 d=null;
  System.out.println(d instanceof Dog2
 }
}
```

Output: true

Method & Constructor overloading

Overloading in Java

- Overloading allows different methods to have same name, but different signatures where signature can differ by number of input parameters or type of input parameters or both.
- Overloading is related to compile time (or static) polymorphism.

Example

```
public class Sum {

    // Overloaded sum(). This sum takes two int parameters
    public int sum(int x, int y)
    {
        return (x + y);
    }

    // Overloaded sum(). This sum takes three int parameters
    public int sum(int x, int y, int z)
    {
        return (x + y + z);
    }

    // Overloaded sum(). This sum takes two double parameters
    public double sum(double x, double y)
    {
        return (x + y);
    }
```

Example

```
// Driver code
    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

What is the advantage?

- We don't have to create and remember different names for functions doing the same thing.
- For example, in our code, if overloading was not supported by Java, we would have to create method names like sum1, sum2, … or sum2Int, sum3Int, … etc.

Can we overload methods on return type?

We cannot overload by return type. This behavior is same in C++. Refer this for details

```
public class Main {
    public int foo() { return 10; }

    // compiler error: foo() is already defined
    public char foo() { return 'a'; }

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

Some Interesting questions:

- **Can we overload static methods?**
- **Can we overload methods that differ only by static keyword?**
- **Can we overload main() in Java?**

Different ways of Method Overloading in Java

- Java can distinguish the methods with different method signatures. i.e. the methods can have same name but with different parameters list (i.e. number of the parameters, order of the parameters, and data types of the parameters) within the same class.

- Overloaded methods are differentiated based on the number and type of the parameters passed as an arguments to the methods.

- You can not define more than one method with the same name, Order and the type of the arguments. It would be compiler error.

- The compiler does not consider the return type while differentiating the overloaded method. But you cannot declare two methods with the same signature and different return type. It will throw a compile time error.

- If both methods have same parameter types, but different return type, then it is not possible. (Java SE 8 Edition)

Why do we need Method Overloading ?

- If we need to do same kind of the operation with different ways i.e. for different inputs. In the example described below, we are doing the addition operation for different inputs. It is hard to find many different meaningful names for single action.

Different ways of doing overloading methods

- The number of parameters in two methods.
- The data types of the parameters of methods.
- The Order of the parameters of methods.

Constructor Overloading

- In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example

```
class Student5{
    int id;
    String name;
    int age;
    //creating two arg constructor
    Student5(int i,String n){
    id = i;
    name = n;
    }
    //creating three arg constructor
    Student5(int i,String n,int a){
    id = i;
    name = n;
    age=a;
    }
    void display(){System.out.println(id+" "+name+" "+age);}
```

Example

```
public static void main(String args[]){
    Student5 s1 = new Student5(111,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
   }
}
```

Output:

111 Karan 0

222 Aryan 25

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

Method Overriding in Java

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.

- If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.

- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- method must have same name as in the parent class
- method must have same parameter as in the parent class.
- must be IS-A relationship (inheritance).

Understanding the problem without method overriding

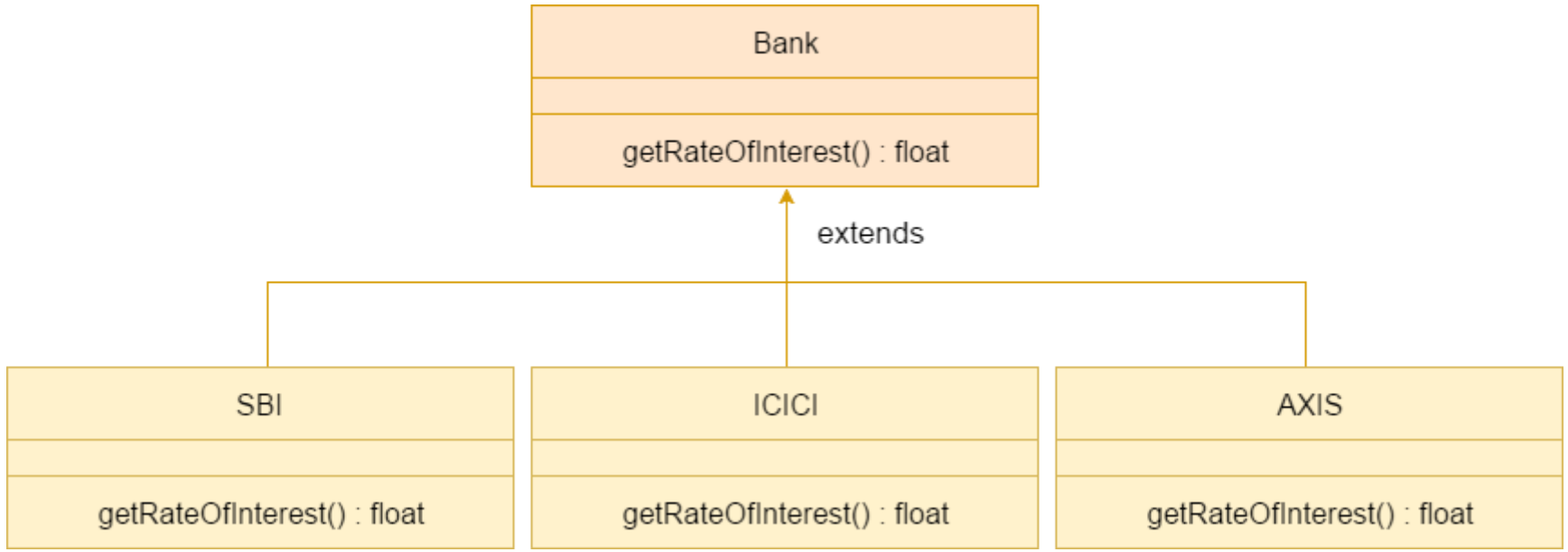
```
class Vehicle{
  void run(){System.out.println("Vehicle is running");}
}
class Bike extends Vehicle{

  public static void main(String args[]){
  Bike obj = new Bike();
  obj.run();
  }
}
```

Output:Vehicle is running

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of Java Method Overriding

- Consider a scenario, Bank is a class that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.



```
class Bank{
int getRateOfInterest(){return 0;}
}

class SBI extends Bank{
int getRateOfInterest(){return 8;}
}

class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}
```

```
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getR
ateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.get
RateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.ge
tRateOfInterest());
}
}
```

Interesting Questions

- Can we override static method?
- Why we cannot override static method?
- Can we override java main method?

@Override annotation

- @Override annotation is used when we override a method in sub class.
- Generally novice developers overlook this feature as it is not mandatory to use this annotation while overriding the method.
- Here we will discuss why we should use @Override annotation and why it is considered as a best practice in java coding.

Example

```
class ParentClass
{
        public void displayMethod(String msg){
                System.out.println(msg);
        }
}
class SubClass extends ParentClass
{
        @Override
        public void displayMethod(String msg){
                System.out.println("Message is: "+ msg);
        }
        public static void main(String args[]){
                SubClass obj = new SubClass();
                obj.displayMethod("Hey!!");
        }
}
```

- In the above example we are overriding a method displaymethod() in the child class.
- Even if we don’t use the @Override annotation, the program would still run fine without any issues, we would be wondering the why do we use this annotation at all.

Why we use @Override annotation

- Using @Override annotation while overriding a method is considered as a best practice for coding in java because of the following two advantages:

- If programmer makes any mistake such as wrong method name, wrong parameter types while overriding, you would get a compile time error. As by using this annotation you instruct compiler that you are overriding this method. If you don't use the annotation then the sub class method would behave as a new method (not the overriding method) in sub class.

- It improves the readability of the code. So if you change the signature of overridden method then all the sub classes that overrides the particular method would throw a compilation error, which would eventually help you to change the signature in the sub classes. If you have lots of classes in your application then this annotation would really help you to identify the classes that require changes when you change the signature of a method.

Final Keyword

Final Keyword

- The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context.

Final can be:

- variable
- method
- class

Final Keyword

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

Interesting Questions

- What is blank or uninitialized final variable?
- Can we initialize blank final variable?
- What us a static blank final variable?
- What is final parameter?
- Can we declare a constructor final?

SYNERGETICS GET IT RIGHT | Microsoft Partner
Gold Cloud Platform
Silver Learning

Q & A

Contact: amitmahadik@synergetics-india.com

SYNERGETICS GET IT RIGHT | Microsoft Partner
Gold Cloud Platform
Silver Learning

Thank You