Dynamic Polymorphism

Dynamic Method Dispatch or Runtime Polymorphism

- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs.
- A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.

```
class A
{
   void fun()
   {
      System.out.println("Inside A's  method");
   }
}

class B extends A
{
   // overriding fun()
   void fun()
   {
      System.out.println("Inside B's  method");
   }
}

class C extends A
{
   // overriding fun()
   void fun()
   {
      System.out.println("Inside C's  method");
   }
}
```

```
class Dpolymorphism
{
    public static void main(String args[])
    {
        A a = new A();
        B b = new B();
        C c = new C();
        A ref;
        ref = a;
        ref.fun();
        ref = b;
        ref.fun();
        ref = c;
        ref.fun();
    }
}
```

Runtime Polymorphism with Data Members

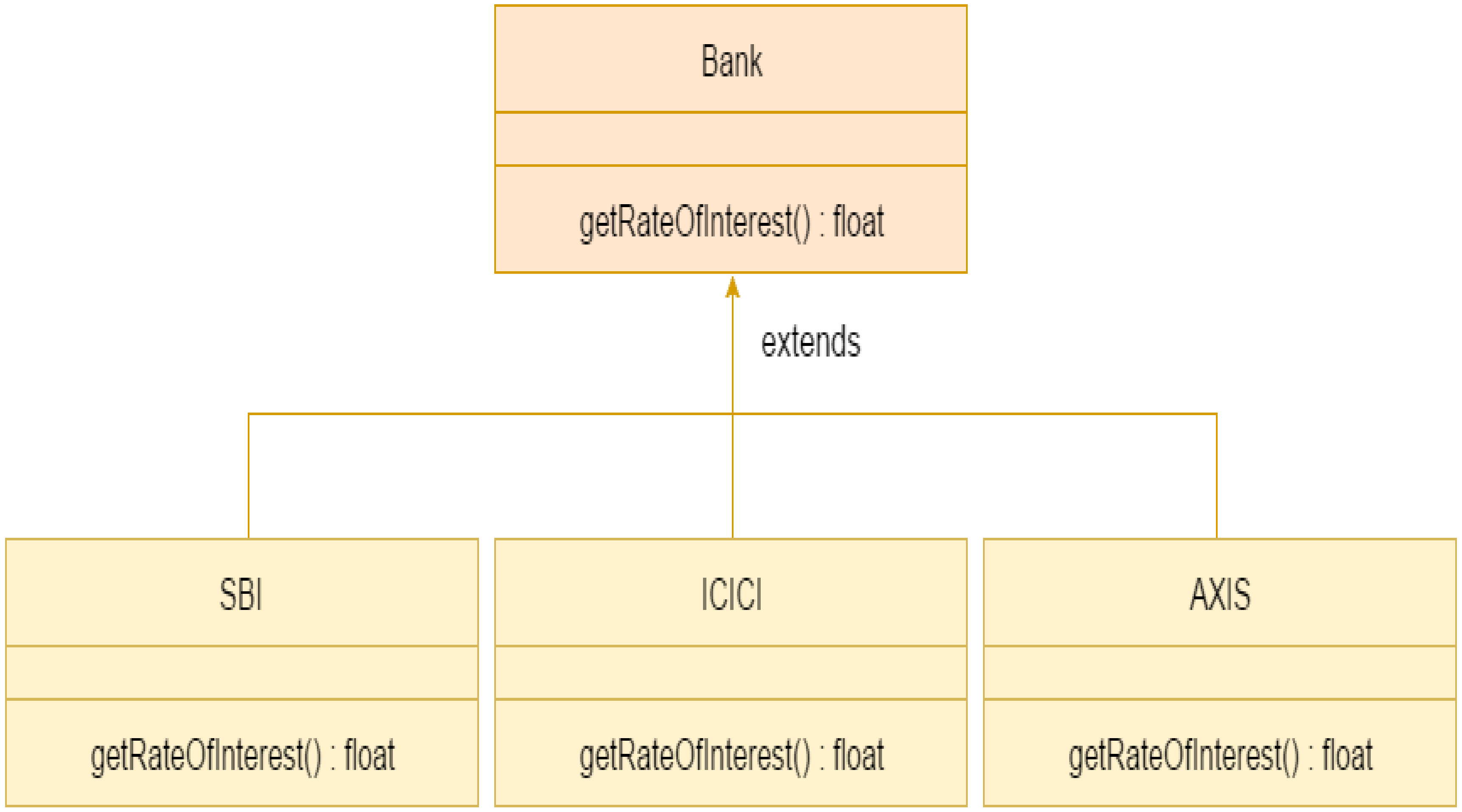
- In Java, we can override methods only, not the variables(data members), so **runtime polymorphism cannot be achieved by data members.**

```
class A
{
   int x = 10;
}
class B extends A
{
   int x = 20;
}
public class Test
{
   public static void main(String args[])
   {
      A a = new B(); // object of type B
      System.out.println(a.x);
   }
}
```

Advantages of Dynamic Method Dispatch

- Dynamic method dispatch allow Java to support overriding of method which is central for run-time polymorphism.
- It allows a class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
- It also allow subclasses to add its specific methods subclasses to define the specific implementation of some.
- private, final and static methods and variables uses static binding and bonded by compiler while overridden methods are bonded during runtime based upon type of runtime object

Java Runtime Polymorphism Example: Bank


Bank
getRateOfInterest() : float
extends
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float

```
class Bank{
        float getRateOfInterest(){return 0;}
}
class SBI extends Bank{
        float getRateOfInterest(){return 8.4f;}
}
class ICICI extends Bank{
        float getRateOfInterest(){return 7.3f;}
}
class AXIS extends Bank{
        float getRateOfInterest(){return 9.7f;}
}
class TestPolymorphism{
    public static void main(String args[]){
        Bank b;
        b=new SBI();
        System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
        b=new ICICI();
        System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
        b=new AXIS();
        System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
     }
}
```

Static method Binding

```
public class ClassA {
    public static void printStatic(){
        System.out.println("hi static from A");
    }
    public void printDynamic(){
        System.out.println("hi dynamic from A");
    }
}
public class ClassB extends ClassA {
    public static void printStatic(){
        System.out.println("hi static from B");
    }
    public void printDynamic(){
        System.out.println("hi dynamic from B");
    }   }
```

```
public static void main (String args[]){
    ClassA x1 = new ClassB();
    x1.printStatic();
    x1.printDynamic();
    ClassB x2 = new ClassB();
    x2.printStatic();
    x2.printDynamic();
}
```

Final Keyword In Java

- 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
- **1) Java final variable**
- If you make any variable as final, you cannot change the value of final variable(It will be constant).

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

Output: compile time error

2) Java final method

- If you make any method as final, you cannot **override** it.

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

class Honda extends Bike{
  void run(){System.out.println("running safely with 100kmph")
    ;}

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

3) Java final class

- If you make any class as final, you **cannot extend** it.

```
final class Bike{}
  class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}


}
  public static void main(String args[]){
  Honda1 honda= new Honda();
  honda.run();
  }
}
```

Q) Is final method inherited?

- Ans) Yes, final method is inherited but you cannot override it.

```
class Bike{
 final void run(){System.out.println("running...");}
}
class Honda2 extends Bike{
  public static void main(String args[]){
   new Honda2().run();
  }
}
```

Q) What is blank or uninitialized final variable?

```
class Student{
int id;
String name;
final String PAN_CARD_NUMBER;
...
}
```

•

Q) Can we initialize blank final variable?

- Yes, but only in constructor.

```
class Bike10{
 final int speedlimit;//blank final variable

 Bike10(){
 speedlimit=70;
 System.out.println(speedlimit);
 }

 public static void main(String args[]){
   new Bike10();
}
}
```

Q) What is final parameter?

- If you declare any parameter as final, you cannot change the value of it.

```
class Bike11{
 int cube(final int n){
  n=n+2;//can't be changed as n is final
  n*n*n;
 }
 public static void main(String args[]){
   Bike11 b=new Bike11();
   b.cube(5);
 }
}
```

Q) Can we declare a constructor final?

- No, because constructor is never inherited.