**Runtime Polymorphism in Java**

1. [Runtime Polymorphism](http://www.javatpoint.com/runtime-polymorphism-in-java)
2. [Upcasting](http://www.javatpoint.com/runtime-polymorphism-in-java#upcasting)
3. [Example of Runtime Polymorphism](http://www.javatpoint.com/runtime-polymorphism-in-java#polymorphismex)
4. [Runtime Polymorphism with data members](http://www.javatpoint.com/runtime-polymorphism-in-java#polymorphismdatamember)

**Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

**Upcasting**

When **reference variable of Parent class** refers to **the object of Child class**, it is known as upcasting. For example:



1. class A{}
2. class B extends A{}
3. **A a=new B();**//upcasting

**Example of Runtime Polymorphism**

In this example, we are creating two classes Bike and Splendar. Splendar class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

1. class Bike{
2. void run(){System.out.println("running");}
3. }
4. class Splender extends Bike{
5. void run(){System.out.println("running safely with 60km");}
6.
7. public static void main(String args[]){
8. **Bike b = new Splender();//**upcasting
9. b.run();
10. }
11. }

Output:running safely with 60km.

**Real example of Java Runtime Polymorphism**

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



Note: It is also given in method overriding but there was no upcasting.

1. class Bank{
2. int getRateOfInterest(){return 0;}
3. }
4.
5. class SBI extends Bank{
6. int getRateOfInterest(){return 8;}
7. }
8.
9. class ICICI extends Bank{
10. int getRateOfInterest(){return 7;}
11. }
12. class AXIS extends Bank{
13. int getRateOfInterest(){return 9;}
14. }
15.
16. class Test{
17. public static void main(String args[]){
18. Bank b1=new SBI();
19. Bank b2=new ICICI();
20. Bank b3=new AXIS();
21. System.out.println("SBI Rate of Interest: "+b1.getRateOfInterest());
22. System.out.println("ICICI Rate of Interest: "+b2.getRateOfInterest());
23. System.out.println("AXIS Rate of Interest: "+b3.getRateOfInterest());
24. }
25. }

Output:

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

**Runtime Polymorphism with data member**

|  |
| --- |
| Method is overridden not the datamembers, so runtime polymorphism can't be achieved by data members.  |
| In the example given below, both the classes have a datamember speedlimit, we are accessing the datamember by the reference variable of Parent class which refers to the subclass object. Since we are accessing the datamember which is not overridden, hence it will access the datamember of Parent class always.  |

**Rule: Runtime polymorphism can't be achieved by data members.**

1. class Bike{
2. int speedlimit=90;
3. }
4. class Honda extends Bike{
5. int speedlimit=150;
6.
7. public static void main(String args[]){
8. Bike obj=new Honda();
9. System.out.println(obj.speedlimit);//90
10. }

Output:90

**Runtime Polymorphism with Multilevel Inheritance**

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

1. class Animal{
2. void eat(){System.out.println("eating");}
3. }
4.
5. class Dog extends Animal{
6. void eat(){System.out.println("eating fruits");}
7. }
8.
9. class BabyDog extends Dog{
10. void eat(){System.out.println("drinking milk");}
11.
12. public static void main(String args[]){
13. Animal a1,a2,a3;
14. a1=new Animal();
15. a2=new Dog();
16. a3=new BabyDog();
17.
18. a1.eat();
19. a2.eat();
20. a3.eat();
21. }
22. }

Output: eating

 eating fruits

 drinking Milk

**Try for Output**

1. class Animal{
2. void eat(){System.out.println("animal is eating...");}
3. }
4.
5. class Dog extends Animal{
6. void eat(){System.out.println("dog is eating...");}
7. }
8.
9. class BabyDog extends Dog{
10. public static void main(String args[]){
11. Animal a=new BabyDog();
12. a.eat();
13. }}

Output: Dog is eating

Since, BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked.

**Static Binding and Dynamic Binding**



Connecting **a method call** to the **method body** is known as binding.

There are two types of binding

1. static binding (also known as early binding).
2. dynamic binding (also known as late binding).

**Understanding Type**

Let's understand the type of instance.

**1) variables have a type**

Each variable has a type, it may be primitive and non-primitive.

1. int data=30;

Here data variable is a type of int.

**2) References have a type**

1. class Dog{
2. public static void main(String args[]){
3. Dog d1;//Here d1 is a type of Dog
4. }
5. }

**3) Objects have a type**

|  |
| --- |
| An object is an instance of particular java class,but it is also an instance of its superclass. |

1. class Animal{}
2.
3. class Dog extends Animal{
4. public static void main(String args[]){
5. Dog d1=new Dog();
6. }
7. }

|  |
| --- |
| Here d1 is an instance of Dog class, but it is also an instance of Animal. |

**Static binding**

When type of the object is determined at compiled time(by the compiler), it is known as **static binding.**

If there is any private, final or static method in a class, there is static binding.

**Example of static binding**

1. class Dog{
2. private void eat(){System.out.println("dog is eating...");}
3.
4. public static void main(String args[]){
5. Dog d1=new Dog();
6. d1.eat();
7. }
8. }

**Dynamic binding**

When type of the object is determined at run-time, it is known **as dynamic binding.**

**Example of dynamic binding**

1. class Animal{
2. void eat(){System.out.println("animal is eating...");}
3. }
4.
5. class Dog extends Animal{
6. void eat(){System.out.println("dog is eating...");}
7.
8. public static void main(String args[]){
9. **Animal a=new Dog();  //upcasting: reference variable of Parent class** refers to **the object of Child class**
10. a.eat();
11. }
12. }

Output:dog is eating...

|  |
| --- |
| In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type. |

**What is runtime polymorphism in Java?**

Polymorphism is the capability of an action or method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementation. This is one of the basic principles of object oriented programming.

The method overriding is an example of runtime polymorphism. You can have a method in subclass overrides the method in its super classes with the same name and signature. Java virtual machine determines the proper method to call at the runtime, not at the compile time.

Let's take a look at the following example:

class Animal {
 void whoAmI() {
 System.out.println("I am a generic Animal.");
 }
}
class Dog extends Animal {
 void whoAmI() {
 System.out.println("I am a Dog.");
 }
}
class Cow extends Animal {
 void whoAmI() {
 System.out.println("I am a Cow.");
 }
}
class Snake extends Animal {
 void whoAmI() {
 System.out.println("I am a Snake.");
 }
}

class RuntimePolymorphismDemo {

 public static void main(String[] args) {
 Animal ref1 = new Animal();
 Animal ref2 = new Dog();
 Animal ref3 = new Cow();
 Animal ref4 = new Snake();
 ref1.whoAmI();
 ref2.whoAmI();
 ref3.whoAmI();
 ref4.whoAmI();
 }
}

The output is

I am a generic Animal.
I am a Dog.
I am a Cow.
I am a Snake.

In the example, there are four variables of type Animal (e.g., *ref1*, *ref2*, *ref3*, and *ref4*). Only*ref1* refers to an instance of *Animal* class, all others refer to an instance of the subclasses of*Animal*. From the output results, you can confirm that version of a method is invoked based on the actually object's type.

In Java, a variable declared type of class *A* can hold a reference to an object of class *A* or an object belonging to any subclasses of class *A*. The program is able to resolve the correct method related to the subclass object at runtime. This is called the runtime polymorphism in Java. This provides the ability to override functionality already available in the class hierarchy tree. At runtime, which version of the method will be invoked is based on the type of actual object stored in that reference variable and not on the type of the reference variable.

*The Java Zone is brought to you in partnership with [ZeroTurnaround](https://dzone.com/go?i=84821&u=http%3A%2F%2Fpages.zeroturnaround.com%2FRocket-Powered-White-Paper_Rocket-Powered-White-Paper.html%3Futm_source%3Ddzone%26utm_medium%3Dsponsored_link%26utm_campaign%3Drocketpowered). Check out this*[*8-step guide*](https://dzone.com/go?i=84821&u=http%3A%2F%2Fpages.zeroturnaround.com%2FRocket-Powered-White-Paper_Rocket-Powered-White-Paper.html%3Futm_source%3Ddzone%26utm_medium%3Dsponsored_link%26utm_campaign%3Drocketpowered)*to see how you can increase your productivity by skipping slow application redeploys and by implementing application profiling, as you code!*

Quite a long time back I had written about [Overriding v/s Hiding](http://blog.sanaulla.info/2008/02/29/overriding-vs-hiding/). In this post I would like to explain in brief with examples about Runtime polymorphism in Java. This post should have been written before [Overriding v/s Hiding](http://blog.sanaulla.info/2008/02/29/overriding-vs-hiding/), but better late than never.

Let us consider the following Vehicle and Car and Truck class:

class Vehicle{

 public void drive(){

 System.out.println("Driving vehicle ...");

 }

}

class Car extends Vehicle{

 @Override

 public void drive(){

 System.out.println("Driving car...");

 }

}

class Truck extends Vehicle{

 @Override

 public void drive(){

 System.out.println("Driving truck...");

 }

 public void load(){

 System.out.println("Loading truck...");

 }

}

A Vehicle can be driven, so is a Car and Truck. But in addition to this a Truck can also be loaded with goods. Let us create instances of these classes and drive() them and try to also load() the truck.

public class RunTimePolymorphismDemo {

 public static void main(String[] args) {

 Vehicle vehicle = new Vehicle();

 vehicle.drive();

 Vehicle carVehicle = new Car();

 carVehicle.drive();

 Vehicle truckVehicle = new Truck();

 truckVehicle.drive();

 //Compile time error

 //truckVehicle.load();

 Truck truck = new Truck();

 truck.load();

 }

}

And the output is:

Driving vehicle ...

Driving car...

Driving truck...

Loading truck...

Had the runtime polymorphism not kicked in, the output would have been: Driving vehicle ... for all the three invocations of drive() method. You can also see that truckVehicle.drive() results in a compile time error. So what’s happening in the above code?

Any object declaration and instantiation has 2 parts in it: The type of the reference and the type of the object created. For example in Vehicle carVehicle = new Car() the reference type is Vehicle and the object created is of type Car. Such an assignment is only possible when the object created type is a subclass of the reference type i.e in cases where inheritance is used.

Each object reference can be used to invoke methods and the methods which can be invoked is decided based on the reference type. And this is decided during the compile time. But the implementation to be invoked is decided based on the type of the object created. In the above example: carVehicle.drive() compiles because the drive() method is part of the Vehicle class and gives Driving car... as the output because the method is overridden by the Car class. On similar lines: truckVehicle.load() gives compile time error because the method load() is not part of theVehicle class, but is defined only in the Truck class. But the truck.load() compiles because the reference type is Truck class and the compiler can resolve the load() method.

To summarise:

* The method binding happens at the compile time i.e which methods can be invoked on a given reference type is decided at the compile time.
* The selection of the method’s implementation to execute happens at the run time i.e which implementation of the method to be executed i.e the super class version or one of the subclass’s version is decided at the run time and this is what leads to the runtime polymorphism.

There are plenty of places where runtime polymorphism is leveraged, few which I can state: Dependency Injection, [Coding to Interface](http://blog.sanaulla.info/2010/08/26/how-coding-to-interface-and-inheritance-helped-me-reuse-code/).