FAST

National University of Computer and Emerging Sciences Peshawar

Lecture # 15

Software Construction and Development (Java Programming)

Instructor: Muhammad Abdullah Orakzai

DEPARTMENT OF COMPUTER SCIENCE

الذی علم بالقلم۔ علم الانسان ما لم یعلم۔

Polymorphism in Java

Contents

Polymorphism in Java

- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*.
- When one task is performed by different ways then it is known as polymorphism.
- **Eg:** To convince customer differently.

Polymorphism in Java…

- Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means “**many”** and "morphs" means “**forms**”. So polymorphism means many forms.

- There are two types of polymorphism in Java: **compile-time polymorphism** and **runtime polymorphism**. We can perform polymorphism in java by method overloading and method overriding.

Polymorphism in Java…

- If you overload a static method in Java, it is the example of compile time polymorphism.

- Here, we will focus on runtime polymorphism in java.

- **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.

Runtime Polymorphism in Java

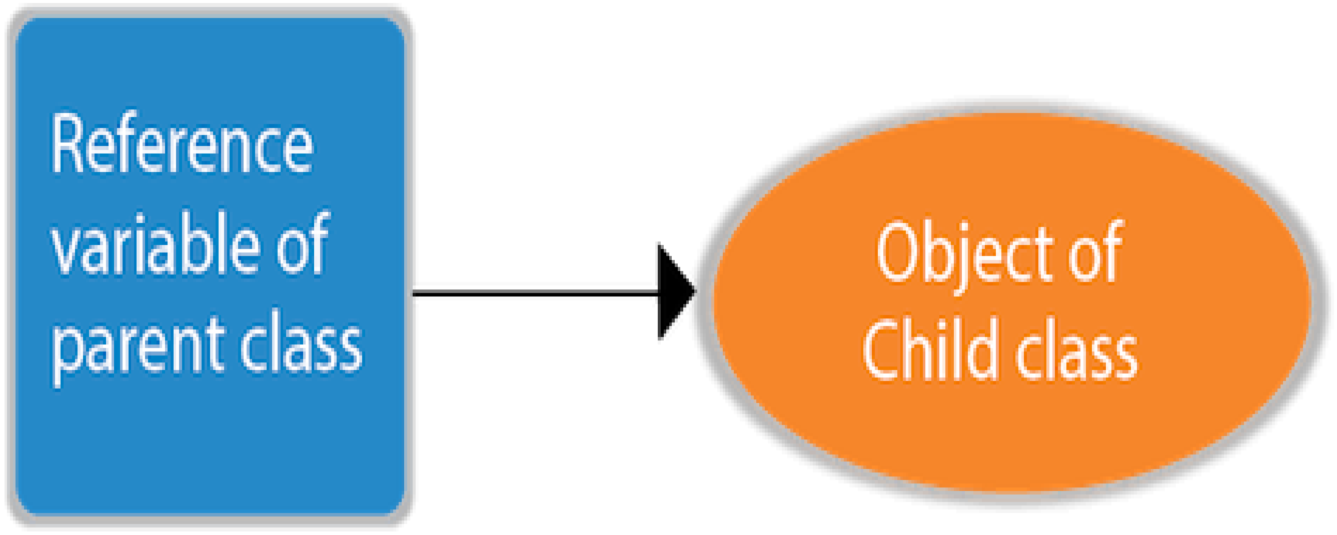
- In this process, an overridden method is called through the reference variable of a superclass.

- 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

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting.



Upcasting…

For example:

```
class A
{


}
class B extends A{}
 A a=new B();      //upcasting
```

Example of Java Runtime Polymorphism

- In this example, we are creating two classes Bike and Splendor. Splendor 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, the subclass method is invoked at runtime.
- Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

Example of Java Runtime Polymorphism…

```
class Bike{
  void run(){System.out.println("running");}
}
class Splendor extends Bike{
  void run(){System.out.println("running safely with 60km");}
  public static void main(String args[]){
    Bike b = new Splendor();                    //upcasting
    b.run();
  }
}
```

Output:

```
running safely with 60km.
```

Java Runtime Polymorphism with Data Member

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

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

Java Runtime Polymorphism with Data Member…

```
class Bike{
 int speedlimit=90;
}
class Honda3 extends Bike{
 int speedlimit=150;
 public static void main(String args[]){
  Bike obj=new Honda3();
  System.out.println(obj.speedlimit);     //90
}
}
```

Output:

```
90
```

Java Runtime Polymorphism with Multilevel Inheritance

```
class Animal{
void eat()   {
System.out.println(“Animal is eating");
 }
}
class Dog extends Animal{
void eat()  {   System.out.println(“Dog eating fruits");   }
}
```

Java Runtime Polymorphism with Multilevel Inheritance…

```
class BabyDog extends Dog{
void eat()   {
System.out.println(“Baby dog drinking milk");
}
public static void main(String args[]){
 Animal a1,a2,a3;      // Reference variables of animal
```

Java Runtime Polymorphism with Multilevel Inheritance…

```
a1=new Animal();

a2=new Dog();

a3=new BabyDog();

a1.eat();

a2.eat();

a3.eat();

}

}
```

Output:

```
Animal is eating
Dog eating fruits

Baby dog drinking Milk
```

Try for Output

```
class Animal{
void eat()   { System.out.println("animal is eating...");  }
}
class Dog extends Animal{
void eat()     {   System.out.println("dog is eating...");   }
}
class BabyDog1 extends Dog{
public static void main(String args[]){
Animal a=new BabyDog1();
a.eat();
}}
```

Output:

```
dog is eating
```

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

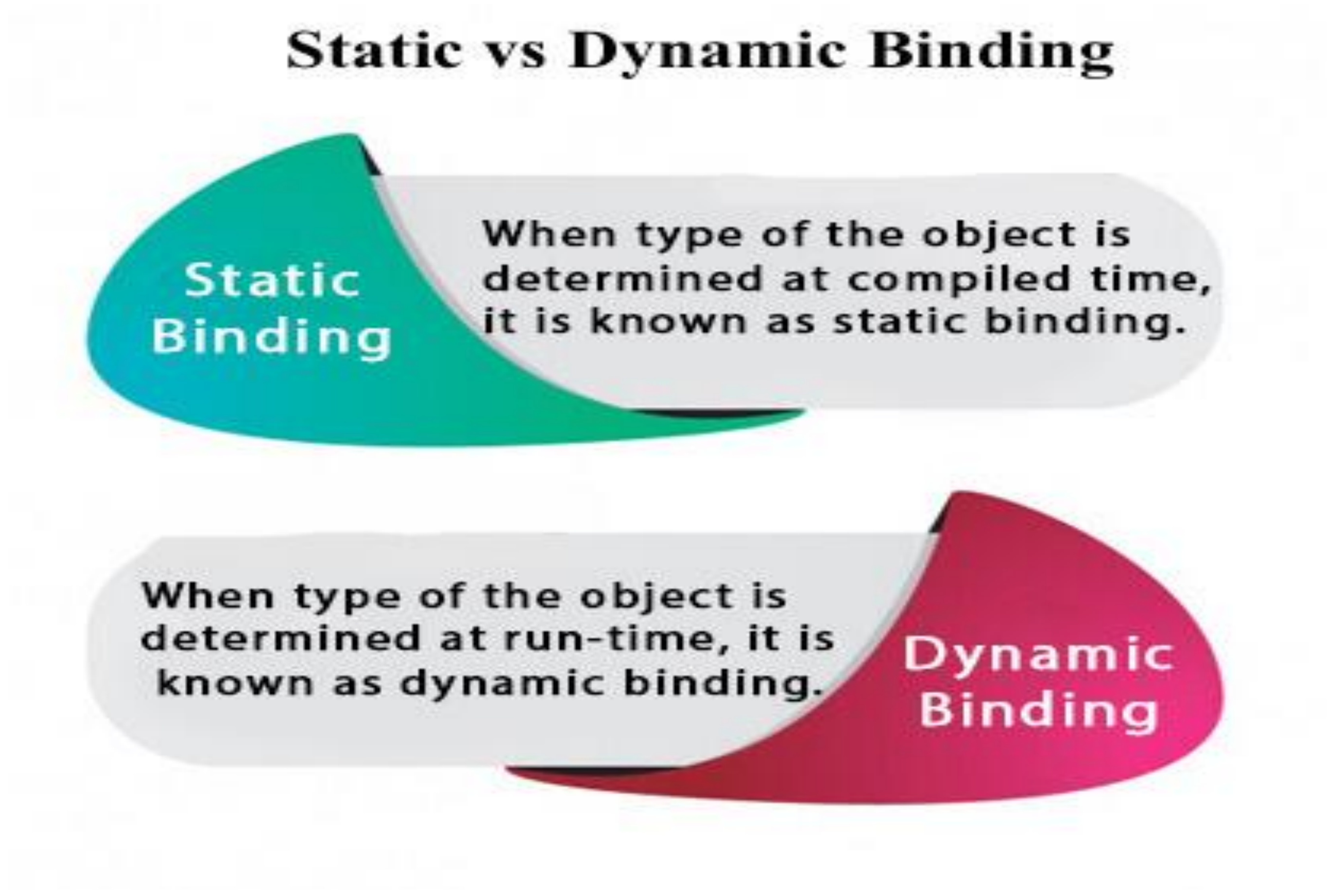
Binding in Java

- ❖Connecting a method call to the method body is known as binding.
- ❖Association of 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).

Binding in Java...


Static vs Dynamic Binding

Static Binding

When type of the object is determined at compiled time, it is known as static binding.

Dynamic Binding

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

Binding in Java…

Let's understand the type of instance.

1) variables have a type

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

int data=30;

Here data variable is a type of int.

Binding in Java…

2) References have a type

```
class Dog{
 public static void main(String args[]){
 Dog d1;                //Here d1 is a type of Dog
 }
}
```

Binding in Java...

3) Objects have a type

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

```
class Animal{}

class Dog extends Animal{
 public static void main(String args[]){
  Dog d1=new Dog();
 }
}
```

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

1) 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.

1) Static Binding…

Example of static binding

```
class Dog{
 private void eat(){System.out.println("dog is eating...");}
 public static void main(String args[]){
  Dog d1=new Dog();
  d1.eat();
 }
}
```

Output:

```
dog is eating…
```

2) Dynamic Binding

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

Example of dynamic binding

```
class Animal{
 void eat()   {
 System.out.println("animal is eating...");
 }
}
```

2) Dynamic Binding…

```
class Dog extends Animal{
 void eat()    {    System.out.println("dog is eating...");    }
  public static void main(String args[]){
  Animal a=new Dog();
  a.eat();
 } }
```

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.

Java instanceof

- 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.

Simple example of java instanceof

Let's see the simple example of instance operator where it tests the current class.

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

Output:

```
true
```

Another example of java instanceof operator

An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.

```
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
```

instanceof in java with a variable that have null value

If we apply instanceof operator with a variable that have null value, it returns false. Let's see the example given below where we apply instanceof operator with the variable that have null value.

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

```
Output: false
```

Downcasting with java instanceof operator

- When Subclass type refers to the object of Parent class, it is known as downcasting.

- If we perform it directly, compiler gives Compilation error. If you perform it by typecasting, ClassCastException is thrown at runtime.

- But if we use instanceof operator, downcasting is possible.

```
Dog d=new Animal();        //Compilation error
```

Downcasting with java instanceof operator…

- If we perform downcasting by typecasting, ClassCastException is thrown at runtime.

```
Dog d =(Dog)new Animal();
//Compiles successfully but ClassCastException is thrown at runtime
```

Possibility of downcasting with instanceof

Let's see the example, where downcasting is possible by instanceof operator.

```
class Animal { }

class Dog3 extends Animal {

 static void method(Animal a) {

  if(a instanceof Dog3){

     Dog3 d=(Dog3)a;    //downcasting

     System.out.println("ok downcasting performed");

  }

 }
```

Possibility of downcasting with instanceof…

```
public static void main (String [] args) {
   Animal a=new Dog3();
   Dog3.method(a);
 }

}
```

Output: ok downcasting performed

Downcasting without the use of java instanceof

Downcasting can also be performed without the use of instanceof operator as displayed in the following example:

```
class Animal { }
class Dog4 extends Animal {
 static void method(Animal a) {
     Dog4 d=(Dog4)a;//downcasting
     System.out.println("ok downcasting performed");
 }
```

Downcasting without the use of java instanceof…

```
 public static void main (String [] args) {
    Animal a=new Dog4();
    Dog4.method(a);
  }
}
```

Output: ok downcasting performed

Let's take closer look at this, actual object that is referred by a, is an object of Dog class. So if we downcast it, it is fine. But what will happen if we write:

```
Animal a=new Animal();
Dog.method(a);    //Now ClassCastException but not in case of instanceof operator
```

Understanding Real use of instanceof in java

Let's see the real use of instanceof keyword by the example given below.

```
interface Printable{}
class A implements Printable{
public void a(){System.out.println("a method");}
}
class B implements Printable{
public void b(){System.out.println("b method");}
}
```

Understanding Real use of instanceof in java…

```
class Call{

void invoke(Printable p){      //upcasting

if(p instanceof A){

A a=(A)p;       //Downcasting

a.a();

}

if(p instanceof B){

B b=(B)p;      //Downcasting

b.b();

}

  }

}  //end of Call class
```

Understanding Real use of instanceof in java…

```
class Test4{
public static void main(String args[]){
Printable p=new B();
Call c=new Call();
c.invoke(p);
}
}
```

```
Output: b method
```

Type Casting

- In computer science, **type conversion** or **typecasting** refers to changing an entity of one data type into another.

- Type casting can be categorized into two types:

1) **Up-casting**
2) **Down-casting**

1) Up-Casting

- Converting a smaller data type into bigger one
- Implicit - we don’t have to do something special
- No loss of information

Examples of

Primitives

```
    int a = 10;

    double b = a;
```

1) Up-Casting…

Classes

```
 Employee    emp    = new Teacher( );
```

2) Down-Casting

- Converting a bigger data type into smaller one
- Explicit - need to mention
- Possible loss of information

Examples of

Primitives

```
double a = 7.65;
int b = (int) a;
```

2) Down-Casting...

Classes

```
Employee  e   = new Teacher( );     // up-casting


Teacher t  =    (Teacher) e;   // down-casting
```

THANK YOU