CHP-21
Java Overriding

In the previous chapter, we talked about superclasses and subclasses. If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final.
The benefit of overriding is: ability to define a behavior that's specific to the subclass type which means a subclass can implement a parent class method based on its requirement.
In object-oriented terms, overriding means to override the functionality of an existing method.

Example:

Let us look at an example.

```
classAnimal{
public void move(){
System.out.println("Animals can move");
}
}
class Dog extends Animal{
public void move(){
System.out.println("Dogs can walk and run");
}
}
public class TestDog{
public static void main(String args[]){
Animal a =new Animal();// Animal reference and object
Animal b =new Dog();// Animal reference but Dog object
a
.move();// runs the method in Animal class
b.move();//Runs the method in Dog class
}
}
```

This would produce the following result:

CHAPTER


```
Animals can move
Dogs can walk and run
```

In the above example, you can see that the even though **b** is a type of Animal it runs the move method in the Dog class. The reason for this is: In compile time, the check is made on the reference type. However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.
Therefore, in the above example, the program will compile properly since Animal class has the method move. Then, at the runtime, it runs the method specific for that object.
Consider the following example:

```
class Animal{
public void move(){
```

```
System.out.println("Animals can move");
}
}
class Dog extendsAnimal{
public void move(){
System.out.println("Dogs can walk and run");
}
public void bark(){
System.out.println("Dogs can bark");
}
}
public class TestDog{
public static void main(String args[]){
Animal a =new Animal();// Animal reference and object
Animal b =new Dog();// Animal reference but Dog object
a.move();// runs the method in Animal class
b.move();//Runs the method in Dog class
b.bark();
}
}
```

This would produce the following result:

```
TestDog.java:30: cannot find symbol
symbol : method bark()
location:class Animal
b.bark();
^
```

This program will throw a compile time error since b's reference type Animal doesn't have a method by the name of bark.

Rules for method overriding:

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.

- The access level cannot be more restrictive than the overridden method's access level. For example, if the superclass method is declared public, then the overriding method in the subclass cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- Constructors cannot be overridden.

Using the super keyword:

When invoking a superclass version of an overridden method the **super** keyword is used.

```
class Animal{
public void move(){
System.out.println("Animals can move");
}
}
c
lass Dog extends Animal{
public void move(){
```

```
super.move();// invokes the super class method
System.out.println("Dogs can walk and run");
}
}
public class TestDog{
public static void main(String args[]){
Animal b =new Dog();// Animal reference but Dog object
b.move();//Runs the method in Dog class
}
}
```

This would produce the following result:

```
Animals can move
Dogs can walk and run
```

CHP-22
Java Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP, occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

Example:

Let us look at an example.

```
public interfaceVegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above example:

- A Deer IS-A Animal
- A Deer IS-A Vegetarian
- A Deer IS-A Deer
- A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal:

```
Deer d =new Deer();
Animal a = d;
```

CHAPTER


```
Vegetarian v = d;
Object o = d;
```

All the reference variables d,a,v,o refer to the same Deer object in the heap.

Virtual Methods:

In this section, I will show you how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes.

We already have discussed method overriding, where a child class can override a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super

keyword within the overriding method.

```
/* File name : Employee.java */
public class Employee
{
private String name;
private String address;
private int number;
public Employee(String name,String address,int number)
{
System.out.println("Constructing an Employee");
this.name = name;
this.address = address;
this.number = number;
}
public void mailCheck()
{
System.out.println("Mailing a check to "+this.name
+" "+this.address);
}
public String toString()
{
return name +" "+ address +" "+ number;
}
publicString getName()
{
return name;
}
public String getAddress()
{
return address;
}
public void setAddress(String newAddress)
{
address = newAddress;
}
public int getNumber()
{
return number;
}
}
```

Now suppose we extend Employee class as follows:

```
/* File name : Salary.java */
public class Salaryextends Employee
{
private double salary;//Annual salary
```



```
public Salary(String name,String address,int number,double
salary)
{
super(name, address, number);
setSalary(salary);
}
public void mailCheck()
{
System.out.println("Within mailCheck of Salary class ");
System.out.println("Mailing check to "+ getName()
+" with salary "+ salary);
}
public double getSalary()
{
return salary;
}
public void setSalary(double newSalary)
```

```
{
if(newSalary >=0.0)
{
salary = newSalary;
}
}
public double computePay()
{
System.out.println("Computing salary pay for "+ getName());
return salary/52;
}
}
```

Now, you study the following program carefully and try to determine its output:

```
/* File name : VirtualDemo.java */
public class VirtualDemo
{
public static void main(String[] args)
{
Salary s =new Salary("Mohd Mohtashim","Ambehta, UP",
3,3600.00);
Employee e =new Salary("John Adams","Boston, MA",
2,2400.00);
System.out.println("Call mailCheck using Salary reference --");
s.mailCheck();
System.out.println("\n Call mailCheck usingEmployee reference--");
e.mailCheck();
}
}
```

This would produce the following result:

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to MohdMohtashim with salary 3600.0
Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to JohnAdams with salary 2400.0
```


Here, we instantiate two Salary objects, one using a Salary reference s, and the other using an Employee reference e.

While invoking *s.mailCheck()* the compiler sees mailCheck() in the Salary class at compile time, and the JVM invokes mailCheck() in the Salary class at run time.

Invoking mailCheck() on e is quite different because e is an Employee reference. When the compiler sees*e.mailCheck()*, the compiler sees the mailCheck() method in the Employee class.

Here, at compile time, the compiler used mailCheck() in Employee to validate this statement. At run time, however, the JVM invokes mailCheck() in the Salary class.

This behavior is referred to as virtual method invocation, and the methods are referred to as virtual methods. All methods in Java behave in this manner, whereby an overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.