Covariant Return Type

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type. Let's take a simple example:

> Note: If you are beginner to java, skip this topic and return to it after OOPs concepts.

Simple example of Covariant Return Type

FileName: B1.java

```
class A{
A get(){return this;}
}

class B1 extends A{
@Override
B1 get(){return this;}
void message(){System.out.println("welcome to covariant return type");}

public static void main(String args[]){
new B1().get().message();
}
}
```

Test it Now

Output:

```
welcome to covariant return type
```

As you can see in the above example, the return type of the get() method of A class is A but the return type of the get() method of B class is B. Both methods have different return type but it is method overriding. This is known as covariant return type.

Advantages of Covariant Return Type

Following are the advantages of the covariant return type.

1) Covariant return type assists to stay away from the confusing type casts in the class hierarchy and makes the code more usable, readable, and maintainable.

2) In the method overriding, the covariant return type provides the liberty to have more to the point return types.

3) Covariant return type helps in preventing the run-time *ClassCastExceptions* on returns.

Let's take an example to understand the advantages of the covariant return type.

FileName: CovariantExample.java

```
class A1
{
  A1 foo()
  {
    return this;
  }

  void print()
  {
    System.out.println("Inside the class A1");
  }
}


// A2 is the child class of A1
class A2 extends A1
{
  @Override
  A1 foo()
  {
    return this;
  }

  void print()
  {
    System.out.println("Inside the class A2");
  }
}

// A3 is the child class of A2
class A3 extends A2
{
  @Override
  A1 foo()
  {
    return this;
  }

  @Override
  void print()
  {
    System.out.println("Inside the class A3");
  }
}

public class CovariantExample
{
  // main method
  public static void main(String argvs[])
  {
   A1 a1 = new A1();

   // this is ok
   a1.foo().print();

   A2 a2 = new A2();

   // we need to do the type casting to make it
   // more clear to reader about the kind of object created
   ((A2)a2.foo()).print();

   A3 a3 = new A3();

   // doing the type casting
   ((A3)a3.foo()).print();

  }
}
```

Output:

```
Inside the class A1
Inside the class A2
Inside the class A3
```

Explanation: In the above program, class A3 inherits class A2, and class A2 inherits class A1. Thus, A1 is the parent of classes A2 and A3. Hence, any object of classes A2 and A3 is also of type A1. As the return type of the method *foo()* is the same in every class, we do not know the exact type of object the method is actually returning. We can only deduce that returned object will be of type A1, which is the most generic class. We can not say for sure that returned object will be of A2 or A3. It is where we need to do the typecasting to find out the specific type of object returned from the method *foo()*. It not only makes the code verbose; it also requires precision from the programmer to ensure that typecasting is done properly; otherwise, there are fair chances of getting the *ClassCastException*. To exacerbate it, think of a situation where the hierarchical structure goes down to 10 - 15 classes or even more, and in each class, the method *foo()* has the same return type. That is enough to give a nightmare to the reader and writer of the code.

The better way to write the above is:

FileName: CovariantExample.java

```
class A1
{
  A1 foo()
  {
    return this;
  }

  void print()
  {
    System.out.println("Inside the class A1");
  }
}


// A2 is the child class of A1
class A2 extends A1
{
  @Override
  A2 foo()
  {
    return this;
  }

  void print()
  {
    System.out.println("Inside the class A2");
  }
}

// A3 is the child class of A2
class A3 extends A2
{
  @Override
  A3 foo()
  {
    return this;
  }

  @Override
  void print()
  {
    System.out.println("Inside the class A3");
  }
}

public class CovariantExample
{
  // main method
  public static void main(String argvs[])
  {
   A1 a1 = new A1();

   a1.foo().print();

   A2 a2 = new A2();

   a2.foo().print();

   A3 a3 = new A3();

   a3.foo().print();

  }
}
```

Output:

```
Inside the class A1
Inside the class A2
Inside the class A3
```

Explanation: In the above program, no typecasting is needed as the return type is specific. Hence, there is no confusion about knowing the type of object getting returned from the method *foo()*. Also, even if we write the code for the 10 - 15 classes, there would be no confusion regarding the return types of the methods. All this is possible because of the covariant return type.

How is Covariant return types implemented?

Java doesn't allow the return type-based overloading, but JVM always allows return type-based overloading. JVM uses the full signature of a method for lookup/resolution. Full signature means it includes return type in addition to argument types. i.e., a class can have two or more methods differing only by return type. javac uses this fact to implement covariant return types.

Output:

```
The number 1 is not the powerful number.
The number 2 is not the powerful number.
The number 3 is not the powerful number.
The number 4 is the powerful number.
The number 5 is not the powerful number.
The number 6 is not the powerful number.
The number 7 is not the powerful number.
The number 8 is the powerful number.
The number 9 is the powerful number.
The number 10 is not the powerful number.
The number 11 is not the powerful number.
The number 12 is not the powerful number.
The number 13 is not the powerful number.
The number 14 is not the powerful number.
The number 15 is not the powerful number.
The number 16 is the powerful number.
The number 17 is not the powerful number.
The number 18 is not the powerful number.
The number 19 is not the powerful number.
The number 20 is the powerful number.
```

Explanation: For every number from 1 to 20, the method *isPowerfulNo()* is invoked with the help of for-loop. For every number, a vector *primeFactors* is created for storing its prime divisors. Then, we check whether square of every number present in the vector *primeFactors* divides the number or not. If all square of all the number present in the vector *primeFactors* divides the number completely, the number is a powerful number; otherwise, not.