Covariant, Contravariant and Invariant

Covariant would mean that a List<String> is a subtype of List<Object>, contravariant that a List<Object> is a subtype of List<String> and invariant that neither is a subtype of the other, i.e. List<String> and List<Object> are inconvertible types.

**Covariant**: a **Cat[]** is an **Animal[]** , **Contravariant**: an Animal[] is a Cat[]

**Invariant**: an **Animal[]** is not a **Cat[]** and a Cat[] is not an Animal[]

**Covariance**: you are allowed to cast A<C> to A<B>, where C is a subclass of B.

**Covariance is about producers.**

**Contravariance**: you’re allowed to cast A<B> to A<C>, where C is subclass of B

**Contravariance is about consumers**.

**Covariance = narrowing conversion. producers**

**Contravariance = widening conversion. consumers**

**Invariance (in this context) = not convertible.**

**Generics**

This is also possible for Generics:

List<String> aList...

List<? extends Object> **covariantList** = aList;

List<? super String> **contravariantList** = aList;

**Covariance**

**class** Child **extends** Parent {  
 @Override  
 **String** get() {  
 **return new** String(**"Child"**);  
 }  
}

**class** Parent {  
 **Object** get() {  
 **return new** Object();  
 }  
}

**Sub.getSomething is covariant** because it returns a subclass of the return type of Super.getSomething (but fulfills the contract of Super.getSomething()).

Example is given below  
**public class** Test1 {  
 **public static void** main(String[] args) {  
 Parent[] parents = **new** Parent[3];  
 **for** (**int** i = 0; i < 3; i++)  
 parents[i] = **new** Child();  
 **for**( Parent p : parents )  
 System.***out***.println(p);//Prints Child@1b6d3586, Child@1b6d3586 etc  
 }  
}

**Contravariance**

class Super {

void doSomething(**String parameter** )

}

class Sub extends Super {

void doSomething(**Object parameter** )

}

**Sub.doSomething is contravariant** because it takes a parameter of a superclass of the parameter of

Super.doSomething (but, again, fullfills the contract of Super#doSomething)

Arrays Are Covariant but Generics are not

Java, an array of type T[] may contain elements of type T or any subtype of T. For instance:

Number[] numbers = new Number[3];

numbers[0] = newInteger(10);

numbers[1] = newDouble(3.14);

numbers[2] = newByte(0);

**Arrays are covariant, Generics are not**

Object[] arr = new String[10];

Basically, an Object[] is a super type of String[], because Object is a super type of String. This is not true with generics. So, the following declaration is not valid, and won't compile:

List<Object> list = new ArrayList<String>(); // Will not compile.

Reason being, generics are invariant.

**Get/Put Principle**

In summary, we **use covariance when we only intend to take generic values out of a structure**. We **use contravariance when we only intend to put generic values into a structure** and we use an invariant when we intend to do both. The best example I have is the following that copies any kind of numbers from one list into another list. It only *gets* items from the source, and it only *puts* items in the destiny.

Example is given below.

**public class** Test1 {  
**public static void** copy(List<? **extends** Number> source , List<? **super** Number> destn) {  
 **for**(Number num : source)  
 destn.add(num);  
}  
 **public static void** main(String[] args) {  
 List<Integer> intList = Arrays.*asList*(1,2,3,4,5);  
 List<Number> numList = **new** ArrayList<Number>();  
 *copy*(intList,numList);  
  
 **for**( Number num : numList)  
 System.***out***.println(num);  
 }  
}

**Overriding Method is covariant.**

**class** Parent {  
 Object get() {  
 **return new** Object();  
 }  
}  
  
**class** Child **extends** Parent {  
 @Override  
 String get() {  
 **return new** String(**"Child"**);  
 }  
}

**Variance** is about relationships between classes with different generics parameters.

**Covariance example**

**class** Lotus **extends** Flower {  
}

**class** Rose **extends** Flower {  
}

**public class** Flower {  
}

**interface** Garden<T **extends** Flower> {  
 T getFlower();  
}

**class** RoseGarden **implements** Garden<Rose> {  
 @Override  
 **public** Rose getFlower() {  
 **return new** Rose();  
 }  
}

**class** LotusGarden **implements** Garden<Lotus> {  
 @Override  
 **public** Lotus getFlower() {  
 **return new** Lotus();  
 }  
}

***//Use of Covariance*****public static void** check2() {  
 Garden<? **extends** Flower> garden = **new** RoseGarden();  
 Flower flower = garden.getFlower();  
 System.***out***.println(flower);  
}

**Contravariance Example**

**interface** FlowerLover<T **extends** Flower> {  
 **void** use(T flower);  
}

**class** AnyFlowerLover **implements** FlowerLover<Rose> {  
 @Override  
 **public void** use(Rose flower) {  
 System.***out***.println(**"I like all flowers!"**);  
 }  
}

***//Use of Contravariance*****public static void** check1() {  
 FlowerLover<? **super** Flower> person = **new** AnyFlowerLover();  
 person.use(**new** Rose());  
}