Programming Paradigms Tutorials
Variances

Variances

Variance is the correlation of subtyping relationships of complex types and the subtyping relationships of their component types. Scala supports variance annotations of type parameters of generic classes, to allow them to be covariant, contravariant, or invariant if no annotations are used. The use of variance in the type system allows us to make intuitive connections between complex types, whereas the lack of variance can restrict the reuse of a class abstraction.

```
class Foo[+A] // A covariant class
class Bar[-A] // A contravariant class
class Baz[A]  // An invariant class
```

Covariance

A type parameter A of a generic class can be made covariant by using the annotation +A. For some class List[+A], making A covariant implies that for two types A and B where A is a subtype of B, then List[A] is a subtype of List[B]. This allows us to make very useful and intuitive subtyping relationships using generics.

Let’s consider this simple class structure:

```
abstract class Animal {
  def name: String
}
case class Cat(name: String) extends Animal
case class Dog(name: String) extends Animal
```

In the following example, the method printAnimalNames will accept a list of animals as an argument and print their names each on a new line. If List[A] were not covariant, the last two method calls would not compile, which would severely limit the usefulness of the printAnimalNames method.

```
object CovarianceTest extends App {
  def printAnimalNames(animals: List[Animal]): Unit = {
    animals.foreach { animal =>
      println(animal.name)
    }
  }

  val cats: List[Cat] = List(Cat("Whiskers"), Cat("Tom"))
  val dogs: List[Dog] = List(Dog("Fido"), Dog("Rex"))

  printAnimalNames(cats)
  printAnimalNames(dogs)
}
```

Contravariance

A type parameter A of a generic class can be made contravariant by using the annotation -A. This creates a subtyping relationship between the class and its type parameter that is similar, but opposite to what we get with covariance. Consider the example:

```
abstract class Printer[-A] {
  def print(value: A): Unit
}

class AnimalPrinter extends Printer[Animal] {
  def print(animal: Animal): Unit =
    println("The animal's name is: " + animal.name)
}

class CatPrinter extends Printer[Cat] {
  def print(cat: Cat): Unit =
    println("The cat's name is: " + cat.name)
}

object ContravarianceTest extends App {
  val myCat: Cat = Cat("Boots")

  def printMyCat(printer: Printer[Cat]): Unit = {
    printer.print(myCat)
  }

  val catPrinter: Printer[Cat] = new CatPrinter
  val animalPrinter: Printer[Animal] = new AnimalPrinter

  printMyCat(catPrinter)
  printMyCat(animalPrinter)
}
```

Invariance

Generic classes in Scala are invariant by default. This means that they are neither covariant nor contravariant. In the context of the following example, Container class is invariant. A Container[Cat] is not a Container[Animal], nor is the reverse true.

```
class Container[A](value: A) {
  private var _value: A = value
  def getValue: A = _value
  def setValue(value: A): Unit = {
    _value = value
  }
}
```

```
val catContainer: Container[Cat] = new Container(Cat("Felix"))
val animalContainer: Container[Animal] = catContainer
animalContainer.setValue(Dog("Spot"))
val cat: Cat = catContainer.getValue
```

This code will end whit error because we try assignee Dog to a Cat.

Exercises

1. The code:
   ```
   abstract class Sequence[+A] {
        def append(x: Sequence[A]): Sequence[A]
   }
   ```
   Generate error:
   ```
   error: covariant type A occurs in contravariant position
   in type // Sequence[A] of value x
   ```
   Please explain why this error occurs

2. Define a generic invariant copy method for mutable collections.

3. Explain how covariance and invariance works for Arrays in Java.

4. Explain how covariance and invariance works for Collections in Java.