



Módulo Profesional: Big Data Aplicado

Scala

Genéricos

```
class MiLista[A] {  
    //uso del tipo A  
}  
  
class MiMapa [key, value] //tipo genérico clave y otro tipo genérico de valores  
val listaDeEnteros = new MiLista[Int]  
val listaDeCadena = new MiLista[String]
```

La clase MiLista que recibe como parámetro A de tipo Genérico.

Value listaDeEnteros que instancia a la clase MiLista de Enteros.

Value listaDeCadena que instancia a la clase MiLista de Cadena.

Métodos Genéricos

```
//métodos genéricos
object MiLista1 {
    def vacia[A]: MiLista[A] = ??? //de momento nos da igual lo que haga la función
}

val vaciaListaDeEnteros = MiLista1.vacia[Int]
```

Problema de Varianza

La **varianza** describe cómo los tipos genéricos se comportan con respecto a la jerarquía de subtipos.

Scala permite tres tipos principales de varianza:

1. Covarianza (+T)
2. Invarianza (T) –sin signo
3. Contravarianza (-T)

Problema de Varianza

El tipo de varianza que debes usar depende del comportamiento que esperas de tu tipo genérico y de la seguridad de tipo:

- **Covarianza (+A)** cuando quieras que tu tipo genérico sea flexible en términos de asignación a tipos más generales pero asegurándote de no modificar el contenido.
- **Contravarianza (-A)** cuando quieras que tu tipo genérico acepte más tipos específicos, pero no necesitas acceder a los elementos del tipo genérico directamente.
- **Invarianza** cuando quieras máxima seguridad de tus tipos.

1. Ejemplo de Covarianza (+A)

```
// Definimos una clase genérica covariante
class Contenedor[+T](val item: T) {
    def obtener: T = item
}
// Jerarquía de tipos
class Objeto
class Coche extends Objeto
class Bicicleta extends Objeto
object CovarianzaSencilla extends App {
    // Creamos un contenedor para un coche
    val contenedorCoche: Contenedor[Coche] = new Contenedor(new Coche)
    // Debido a la covarianza (+T), podemos tratar el contenedor de coches
    // como un contenedor de objetos
    val contenedorObjeto: Contenedor[Objeto] = contenedorCoche
    // Trabajamos con el contenedor como si fuera de tipo Objeto
    println(contenedorObjeto.obtener.getClass.getSimpleName) // Salida: Coche
```

- **Covarianza (+A)** cuando quieras que tu tipo genérico sea flexible en términos de asignación a tipos más generales pero asegurándote de no modificar el contenido.

2. Invarianza (A)

Una clase genérica es **invariante** cuando no permite ninguna relación de subtipo entre los tipos genéricos.

Es decir, *InvariantList[Gato]* no es un subtipo de *InvariantList[Animal]* ni viceversa, incluso si Gato es un subtipo de Animal.

```
//2. INVARIANCE
class InvariantList[A]
val invariantAnimalList: InvariantList[Animal] = new InvariantList[Animal]
val invariantAnimalList1: InvariantList[Animal] = new InvariantList[Gato]
```

La **invarianza** es útil cuando necesitas asegurarte de que el tipo exacto coincide. Es una elección más restrictiva y segura en términos de tipos.

2. Ejemplo Invarianza (A)

```
// Definimos una clase genérica Caja
class Caja[T](val contenido: T) {
    def obtener: T = contenido
}
// Jerarquía de tipos
class Fruta
class Manzana extends Fruta
class Platano extends Fruta
object InvarianzaSencilla extends App {
    // Creamos una caja para manzanas
    val cajaManzana: Caja[Manzana] = new Caja(new Manzana)
    // La invarianza NO permite asignarla a una caja de frutas
    // val cajaFruta: Caja[Fruta] = cajaManzana // Esto da un ERROR
    // Trabajamos con cada caja en su propio tipo
    println(cajaManzana.obtener.getClass.getSimpleName) // Salida: Manzana
```

3. Contravarianza (-A)

Una clase genérica es **contravariante** cuando invierte la jerarquía de subtipos.

Contravarianza (-A) permite que un contenedor de un tipo más general sea tratado como un contenedor de un tipo más específico.

Es útil cuando estamos trabajando con estructuras que realizan operaciones sobre un tipo general pero reciben un tipo más específico.



3. Ejemplo Contravarianza (-A)

```
// Definimos una jerarquía de clases
class Documento
class Reporte extends Documento
class Factura extends Documento
// Definimos una clase contravariante
class Impresora[-A] {
    def imprimir(documento: A): Unit = println(s"Imprimiendo ${documento.getClass.getSimpleName}")
}
run | debug
object Contravariante extends App {
    // Una impresora que puede imprimir cualquier tipo de documento
    val impresoraGeneral: Impresora[Documento] = new Impresora[Documento]
    // Como es contravariante, podemos asignarla a una impresora específica para reportes
    val impresoraReporte: Impresora[Reporte] = impresoraGeneral
    // Usamos la impresora específica para imprimir un reporte
    impresoraReporte.imprimir(new Reporte)
    // También podemos imprimir facturas porque la impresora original acepta cualquier Documento
    impresoraGeneral.imprimir(new Factura)
}
```

Ejercicio 1

```
/*Ejercicio: Indica si es válido y si es Covarianza, Invarianza o Contravarianza
class Parque[A] // Invariante
class ListaAnimales[+A] // Covariante
class Cuidado[-A] // Contravariante

val parqueDeGatos: Parque[Gato] = new Parque[Gato]
val parqueDeAnimales: Parque[Animal] = parqueDeGatos // ¿Es válido?

val listaDeGatos: ListaAnimales[Gato] = new ListaAnimales[Gato]
val listaDeAnimales: ListaAnimales[Animal] = listaDeGatos // ¿Es válido?

val cuidadoDeAnimales: Cuidado[Animal] = new Cuidado[Animal]
val cuidadoDeGatos: Cuidado[Gato] = cuidadoDeAnimales // ¿Es válido?
*/
|
```

Ejercicio 2

```
/*Ejercicio 2: Biblioteca
 //Jerarquía de clases
 class Contenido
 class Libro extends Contenido
 class Video extends Contenido

 // Clases genéricas
 class ColeccionCovariante[+A]
 class ColeccionInvariante[A]
 class GestorContravariante[-A]
```

```
//PREGUNTA 1 ¿Es válido?
 val colecciónDeLibros: ColecciónCovariante[Libro] = new ColecciónCovariante[Libro]
 val colecciónDeContenido: ColecciónCovariante[Contenido] = colecciónDeLibros
```

Ejercicio 2

```
/*Ejercicio 2: Biblioteca
 //Jerarquía de clases
 class Contenido
 class Libro extends Contenido
 class Video extends Contenido

 // Clases genéricas
 class ColeccionCovariante[+A]
 class ColeccionInvariante[A]
 class GestorContravariante[-A]
```

```
//PREGUNTA 2 ¿Es válido?
val colecciónDeLibros: ColecciónCovariante[Libro] = new ColecciónCovariante[Libro]
colecciónDeLibros.add(new Video)
```

Ejercicio 2

```
/*Ejercicio 2: Biblioteca
 //Jerarquía de clases
 class Contenido
 class Libro extends Contenido
 class Video extends Contenido

 // Clases genéricas
 class ColeccionCovariante[+A]
 class ColeccionInvariante[A]
 class GestorContravariante[-A]
```

```
//PREGUNTA 3 ¿Es válido?
val colecciónDeLibros: ColecciónInvariante[Libro] = new ColecciónInvariante[Libro]
val colecciónDeContenido: ColecciónInvariante[Contenido] = colecciónDeLibros
```

Ejercicio 2

```
/*Ejercicio 2: Biblioteca
 //Jerarquía de clases
 class Contenido
 class Libro extends Contenido
 class Video extends Contenido

 // Clases genéricas
 class ColeccionCovariante[+A]
 class ColeccionInvariante[A]
 class GestorContravariante[-A]
```

```
//PREGUNTA 4 ¿Es válido?
val gestorDeContenido: GestorContravariante[Contenido] = new GestorContravariante[Contenido]
val gestorDeLibros: GestorContravariante[Libro] = gestorDeContenido
```

Ejercicio 2

/*RESPUESTAS*/

Pregunta 1: Sí

Es válido porque ColeccionCovariante permite que un subtipo
(ColeccionCovariante[Libro]) se trate como su supertipo (ColeccionCovariante[Contenido]).

Pregunta 2: No

No es válido porque la covarianza restringe agregar elementos.

Si agregas un Video, rompes la coherencia del tipo.

Pregunta 3: No

No es válido porque ColeccionInvariante no permite ninguna relación de subtipo.

Los tipos deben coincidir exactamente.

Pregunta 4: Sí

Es válido porque la contravarianza invierte la relación de subtipo.

Un gestor de contenido general (GestorContravariante[Contenido])

puede ser tratado como un gestor de libros (GestorContravariante[Libro]).

