**Covariants**

class VendingMachine[+A] { don't worry about implementation yet }

def install(softDrinkVM: **VendingMachine[SoftDrink]**): Unit = {

// Installs the soft drink vending machine

}

The method install accepts a VendingMachine of type SoftDrink or subtypes of SoftDrink (Cola and TonicWater).

This is possible because type parameter A is prefixed with a +. It indicates that A+ is covariant in that parameter.

Alternatively, it can be said that class VendingMachine is covariant in its type parameter A.

// covariant subtyping

install(new VendingMachine[Cola])

install(new VendingMachine[TonicWater])

// invariant

install(new VendingMachine[SoftDrink])

//Compile error ! contravariant subtyping

install(new VendingMachine[Drink])

**Covariant subtyping**

T is a subtype of B . B is SUPER type of T

T <: B

VendingMachine[T] <: VendingMachine[B]

If T is a subtype of B then VendingMachine[T] should be a subtype of VendingMachine[B].

Covariance is most commonly used in producers (types that return something) and immutable types.

class VendingMachine[+A](val currentItem: Option[A], items: List[A]) {

def addAll[B >: A](newItems: List[B]): VendingMachine[B] = new VendingMachine(items ++ newItems)

}

The method def addAll[B >: A](newItems: List[B]): VendingMachine[B] has very useful characteristic, that is, a lower bound makes the method addAll very flexible as seen below.

B >: A  is a lower type bound. It means that B is constrained to be a supertype of A.

**Complex sub domain**

I have the folowing classes

SuperCulo-->NormalSuperCulo-->ExplosiveSuperCulo-->Culazo-->Toto

final case class CulosMagazine[+A <: SuperCulo]( val lindosLocus: List[A], val currentLocu: Option[A] ) {}

1) domain < Culo extends from SuperCulo and it is subclass of ExplosiveSuperCulo> .

It only accepts ExplosiveSuperCulo, Culazo and Toto

val inputsOkotes = List(new ExplosiveSuperCulo("culo explosivo"),new Culazo("this a culazo"), new Toto("this is a terrible toto"))

val culosMagazine: CulosMagazine[ExplosiveSuperCulo] = CulosMagazine(inputsOkotes, Some(inputsOkotes.tail.head))

2) domain < Culo extends from SuperCulo and it is subclass of Culazo>

It only accepts Toto and Culazo. All subclasses of Culazo and extends from SuperCulo.

val inputsOkotes = List(new Culazo("this a culazo"), new Toto("this is a terrible toto"))

val culosMagazine: CulosMagazine[Culazo] = CulosMagazine(inputsOkotes, Some(inputsOkotes.tail.head))

**Function T –to--> R**

class CuloConverter[-T <: SuperCulo, +R](val factory: (String) => R) extends AnyRef {

def convert(locu: T): R = return factory(locu.name)

}

case object CulosFactory {

val normalSuperCulo = { (paramter: String) => new NormalSuperCulo(paramter) }

}

How to use it

==========

val expectedResult = new Culazo("Pan dulce")

val culoConverter = new CuloConverter[Toto,Culazo](CulosFactory.culazo)

val explosiveSuperCulo = culoConverter.convert(new Toto("Pan dulce"))