02. First Scala

Playframework Java , Akka

Scala

.

: https://docs.scala-lang.org/tour/tour-of-scala.html

.

Generic

```
class Stack[A] {
  private var elements: List[A] = Nil
  def push(x: A) { elements = x :: elements }
  def peek: A = elements.head
  def pop(): A = {
    val currentTop = peek
    elements = elements.tail
    currentTop
  }
}

val stack = new Stack[Int]
stack.push(1)
stack.push(2)
println(stack.pop)  // prints 2
println(stack.pop)  // prints 1

class Fruit
class Apple extends Fruit
class Banana extends Fruit

val stack2 = new Stack[Fruit]
val apple = new Apple
val banana = new Banana

stack2.push(apple)
stack2.push(banana)
println(stack2.pop.getClass.getSimpleName)  // prints Banana
println(stack2.pop.getClass.getSimpleName)  // prints Apple
```

OOP - VARIANCES

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


println("# Covariance")
abstract class Animal {
  def name: String
}
case class Cat(name: String) extends Animal
case class Dog(name: String) extends Animal

def printAnimalNames(animals: List[Animal]): Unit = {
  println("##" + animals )
  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)


println("# Contravariance")

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)
}

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)

println("# Invariance")

abstract class SmallAnimal extends Animal
case class Mouse(name: String) extends SmallAnimal
var smallAni:SmallAnimal = new Mouse("miky")

println(smallAni.name)



//val cat: Cat = catContainer.getValue // Oops, we'd end up with a Dog assigned to a Cat
```

OOP - UPPER

```
abstract class Animal {
 def name: String
}

abstract class Pet extends Animal {}

class Cat extends Pet {
  override def name: String = "Cat"
}

class Dog extends Pet {
  override def name: String = "Dog"
}

class Lion extends Animal {
  override def name: String = "Lion"
}

class PetContainer[P <: Pet](p: P) {
  def pet: P = p
}

val dogContainer = new PetContainer[Dog](new Dog)
val catContainer = new PetContainer[Cat](new Cat)


// this would not compile
val lionContainer = new PetContainer[Lion](new Lion)
```

OOP - LOWER

```
/*
trait Node[+B] {
  def prepend(elem: B): Node[B]
}

case class ListNode[+B](h: B, t: Node[B]) extends Node[B] {
  def prepend(elem: B): ListNode[B] = ListNode(elem, this)
  def head: B = h
  def tail: Node[B] = t
}

case class Nil[+B]() extends Node[B] {
  def prepend(elem: B): ListNode[B] = ListNode(elem, this)
}*/

trait Node[+B] {
  def prepend[U >: B](elem: U): Node[U]
}

case class ListNode[+B](h: B, t: Node[B]) extends Node[B] {
  def prepend[U >: B](elem: U): ListNode[U] = ListNode(elem, this)
  def head: B = h
  def tail: Node[B] = t
}

case class Nil[+B]() extends Node[B] {
  def prepend[U >: B](elem: U): ListNode[U] = ListNode(elem, this)
}

trait Bird
case class AfricanSwallow() extends Bird
case class EuropeanSwallow() extends Bird


val africanSwallowList= ListNode[AfricanSwallow](AfricanSwallow(), Nil())
val birdList: Node[Bird] = africanSwallowList
birdList.prepend(new EuropeanSwallow)
```

Unknown macro: 'html'