Polymorphism

Intro

If we talk about **Polymorphism**, we talk about Language Type System. Why it's so? Because Polymorphism is the property of language type system. The type system introduces additional restrictions that help to check the correctness of programs.

In a language with static typing, the values of all variables and expressions must be defined at the compilation stage. This restriction is not always convenient. First of all, when you need to define a type at runtime.

Polymorphism allows us to bypass the limitation of the static type system, without violating the type safety.

Polymorphism is of four kinds, which in general are grouped into two categories:

- Universal
    - parametric polymotphism/generics
    - including polimorphism/subtyping
- Ad-hoc (special)
    - overloading and typeclasses
    - coercion

We will consider **Universal** Polymorphism.

A feature of universally polymorphic functions and types is that they can equally process (potentially) an infinite number of types.

Using parametric polymorphism allows you to create generic functions and data types, specifying instead of actual types. This feature allows you to process values regardless of their type.

Including polymorphism limits the behavior of a polymorphic function by a set of types that are linked by an inheritance hierarchy (supertype-subtype relationships).

It's all about quantification and variance of type constructors. But we will not scribble our heads definitions and see how it works on a simple example.

List Implementation

Let's start with List of Integers implementation

```scala
sealed trait IntList {
  def isEmpty: Boolean
  def head: Int
  def tail: IntList
  override def toString =
    if (isEmpty) "Nil" else s"$head :: ${tail.toString}"
}
case object Nil extends IntList {
  def isEmpty = true
  def head = throw  new NoSuchElementException("Nil.head")
  def tail = throw  new NoSuchElementException("Nil.tail")
}
case class Cons(val head: Int, val tail: IntList) extends IntList {
  def isEmpty = false
}
```

We have base type *IntList*, class-successor *Cons* and object *Nil*.

Let's consider it more carefully.

The *Nil* constructor creates an empty list and since it does not contain elements, access to them causes exceptions.

We will use it as a marker for the end of a non-empty list. The constructor of the non-empty *Cons* list contains the first element and the list of other elements, which can be an empty list as well.

Using the *case* keyword when declaring list constructors, we get some advantages over ordinary classes. For example, case classes can be used in pattern matching, thus allowing the list to be deconstructed.

Also, for case classes, a companion object is created that contains the apply method, which allows you to create instances of the class without the new keyword, and the designer's parameters become public fields of the class. In Scala, the constructor is part of the class definition, so the constructor parameters in the class Cons become public and override the abstract methods of the base class.

Since Nil does not take values as parameters, we define it as a single object - case object.

Now let's consider example.

```scala
val intList: IntList = Cons(1, Cons(2, Cons(3, Nil)))
```

```
scala> val intList: IntList = Cons(1, Cons(2, Cons(3, Nil)))
intList: IntList = 1 :: 2 :: 3 :: Nil
```

Parametric polymotphism

```
sealed abstract trait List[+T] {  // конструктор типа
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
  override def toString =
    if (isEmpty) "Nil" else s"$head :: ${tail.toString}"
}

case object Nil extends List[Nothing] {
  def isEmpty = true
  def head = throw  new NoSuchElementException("Nil.head")
  def tail = throw  new NoSuchElementException("Nil.tail")
}

case class Cons[T](head: T, tail: List[T]) extends List[T] { // конструктор
типа
  def isEmpty = false
}
```

And several examples

```
val intList: List[Int] =
  Cons[Int](1, Cons[Int](2, Cons[Int](3,Nil)))

val stringList: List[String] =
  Cons[String]("uno", Cons[String]("duo", Cons[String]("tres", Nil)))
```

```
scala> val intList: List[Int] =
     |   Cons[Int](1, Cons[Int](2, Cons[Int](3,Nil)))
intList: List[Int] = 1 :: 2 :: 3 :: Nil

scala> val stringList: List[String] =
     |   Cons[String]("uno", Cons[String]("duo", Cons[String]("tres", Nil)))
stringList: List[String] = uno :: duo :: tres :: Nil
```

What about polymorphic methods?

```
  def map[U]( f : T => U ) : List[U] = {
    this match {
      case Cons(head, tail) => Cons(f(head), tail.map(f))
      case Nil => Nil
    }
  }
```

Use case example

```
val intList = Cons(1, Cons(2, Cons(3, Nil)))

val doubleList: List[Double] =
  intList.map[Double]( (x: Int) => { x * 0.1 } )
```

```
scala> val intList: List[Int] = Cons(1, Cons(2, Cons(3, Nil)))
intList: List[Int] = 1 :: 2 :: 3 :: Nil

scala> val doubleList: List[Double] =
     |   intList.map[Double]( (x: Int) => { x * 0.1 } )
doubleList: List[Double] = 0.1 :: 0.2 :: 0.30000000000000004 :: Nil
```

Using Lambda syntax

```
val doubleList = intList.map { _ * 0.1 }
```

What about **+** ?

Let's try without it

```
val l: List[Double] = Cons[Double](0.5, Nil)
```

```
scala> val l: List[Double] = Cons[Double](0.5, Nil)
<console>:11: error: type mismatch;
 found   : Nil.type
 required: List[Double]
Note: Nothing <: Double (and Nil.type <: List[Nothing]), but trait List is
invariant in type T.
You may wish to define T as +T instead. (SLS 4.5)
       val l: List[Double] = Cons[Double](0.5, Nil)
```

When you create an instance of the Cons class, the Double type is specified as the actual type parameter, which means that the constructor of the Cons class expects that it will pass two values with the types Double and List [Double], respectively.

As the first value, we pass 0.5, which corresponds to the expected Double type, but as the second value, we pass an instance of type List [Nothing] and this is not the type that the compiler expects.

Although Nothing is a subtype of Double, we can not say the same about the types List [Double] and List [Nothing]. In Scala, all parameterized types are invariant by default. In order for List [Nothing] to be considered as a valid argument, where List [Double] is expected, we should change the variability of the polymorphic List type by adding the + sign to the parameter.

Using the type Nothing and declaring a list contravariant to its type parameter is an example where parametric polymorphism and inclusion polymorphism complement each other.

Bad trip

But sometimes the variability check prevents you from implementing, in principle, the correct things. For example, let's add to our list a method for adding a new element

```
def ::(hd: T): List[T] = Cons(hd, this)
```

Be careful here

Unfortunately, the above code is not compiled and the compiler will swear that the covariant type T is in the contravariant position, that is, as the type of the formal parameter hd.

```
def ::[B](hd: B): List[B] = Cons(hd, this)
```

Now the method :: has its own parameter of type B, and since it is invariant (invariant types can appear in any position), its appearance as a formal parameter type of the function should not cause problems. However, the problem appears in the implementation of this method. Now, we can not construct a new list by adding an element of type B to a list of elements of type T. And this is logically true, because a parameter of type B can represent any type. Now, if it could be somehow limited to only valid types. This is done as follows:

```
def :: B >: T](hd: B): List[B] = Cons(hd, this)
```

That is, we added>: T to the type parameter, thereby indicating that the type T is the lower bound of type B (B must be a super-type for T). And with this condition, the code is compiled successfully. Also, thanks to the addition of the type parameter to the :: method, we have further improved its implementation. If we add an element with a type that is different from the original list type, the resulting list will change its type to the type of the new element.

Use case exaple.

```
val strings = "one" :: "two" :: "three" :: Nil
val anyRefString = AnyRef :: strings
```

```
scala> val strings = "one" :: "two" :: "three" :: Nil
strings: List[String] = one :: two :: three :: Nil

scala> val anyRefString = AnyRef :: strings
anyRefString: List[Object] = object AnyRef :: one :: two :: three :: Nil
```

Notice how the new list is constructed on the first line. Such a record is possible due to the right associativity of the :: method. If you rewrite this expression in a more familiar form, it will look like this

```
Nil.::("three").::("two").::("one")
```

In addition to specifying the lower bound, it is also possible to indicate the upper boundary of the type T <: U:

```
scala> def personName[T <: Person](p: T): String = p.name
personName: [T <: Person](p: T)String

scala> def studentName[T <: Student](s: T): String = s.name
studentName: [T <: Student](s: T)String

scala> val students: List[Student] = Student("Bob") :: Student("Joe") :: Nil
students: List[Student] = List(Bob, Joe)

scala> val persons: List[Person] = students
persons: List[Person] = List(Bob, Joe)

scala> students map personName
res0: List[String] = List(Bob, Joe)

scala> students map studentName
res1: List[String] = List(Bob, Joe)

scala> persons map studentName
<console>:14: error: type mismatch;
 found   : Student => String
 required: Person => ?
              persons map studentName
```