Types and pattern matching

Pure Object Oriented Language

- In a pure object oriented language, every value is an object

- If based on classes, then the type of each value is a class

- Is Scala a pure object oriented language?
  - What about functions?
  - What about primitives?

- Is Java? (No)

How are primitives treated

- Primitive types like Int or Boolean, treated like any other class

- Compiler represents scala.Int to JVM’s primitive type - 32-bit int, scala.Boolean to JVM’s primitive type boolean etc

Example

Let’s look at how natural numbers can be constructed as objects, ie 0, 1, 2, 3, …

```
types_and_pattern_matching.naturals
```

Functions as objects

- Function values are treated as objects in Scala
- Function type A=>B is abbreviation for the trait scala.Function1[A,B]
  - This has the apply method: Class[_ <: Int => Int]
- Functions are objects with an apply method
  - Traits Function2, etc for multiple parameters
  - See more: http://www.scala-lang.org/api/2.9.2/scala/Function1.html

```
scala> def next(n:Int):Int = n+1
next: (n: Int)Int
scala> next _ getClass
res3: Class[_ <: Int => Int] = class $$Lambda$1123/1125499532
```

Expansion behind the scenes

- What does the expansion of an anonymous function look like ?

```
(x: Int) => x * x
// expands into


{ class AnonFun extends Function1[Int, Int] {
    def apply(x: Int) = x * x
  }
  new AnonFun
}
```

Anonymous class syntax

- Anonymous classes are unnamed classes
- How can we create an instance?
- By using the reserved word new and defining the body with braces
- Also, instances can be created from traits
  - This is syntax sugar.

```
//anonymous class
val myPoint = new{ val x = 1; val y = 2 }

// instance from trait
trait AnonymousHero {
  def superpower: String
}

val myHero = new AnonymousHero {
  def superpower = "I can compile Scala with my brain"
}
```

Expansion, even shorter

- Using the anonymous class syntax, the expanded function is even shorter

```
(x: Int) => x * x
// expands into

new Function1[Int, Int] {
  def apply(x: Int) = x * x
}
```

Methods and functions

- In many situations, you can ignore the difference between functions and methods
- But, methods such as def f(x: Int): Boolean = … are not functions
    - A Scala method, as in Java, is a part of a class. It has a name, a signature, optionally some annotations, and some bytecode
    - A function in Scala is a complete object. There are a series of traits in Scala to represent functions with various numbers of arguments: Function0, Function1
- A method is converted to a function value, if it's name is used in a place, where a function value is expected
    - When we treat a method as a function, such as by assigning it to a variable, Scala actually creates a function object whose apply method calls the original method, and that is the object that gets assigned to the variable

```
def f(x: Int): Boolean = …
// is converted to
(x: Int) => f(x)
// in expanded form
new Function1[Int, Boolean] {
  def apply(x: Int) = f(x)
}
```

Example

- List revisited - how to use the function parameter syntax on an object?

```
polymorphism.List
```

OO + FP Polymorphism

- OO - subtyping
- FP - generics
- Let us look at their interactions
  - bounds - constraints for type parameters
  - variance - how parametrized types behave when subtyped

Type Bounds

- Let us assume that a method returns a value, but can also throw an exception

- What is the most precise return type ?

```
def test( list: List): Boolean
```

Upper type bounds

- type parameters and abstract types may be constrained by a type bound
- An upper type bound **T <: A** declares that type variable T refers to a subtype of type A
- Type bound let us assume some knowledge about the paremetrized type
  - Without the upper type bound annotation, it would not be possible to call method isSimilar in method findSimilar.

```
trait Similar {
  def isSimilar(x: Any): Boolean
}
object UpperBoundTest extends Application {
  def findSimilar[T <: Similar](e: T, xs: List[T]): Boolean =
    if (xs.isEmpty) false
    else if (e.isSimilar(xs.head)) true
    else findSimilar[T](e, xs.tail)
}
```

Lower type bounds

- Lower type bounds declare a type to be a supertype of another type

- The term **T >: A** expresses that the type parameter **T** or the abstract type **T** refer to a supertype of type **A**.

```
[S >: NonEmpty]
```

Mixed bounds

- It is possible to mix a lower bound with an upper bound
- We could express, that any type between NonEmpty and IntSet are allowed

```
[ S >: NonEmpty <:IntSet]
```

Covariance

- Int is a subtype of object : Int <: Object
- How about List[Int] <: List[Object]

Lab Time!

types_and_pattern_matching_08