Week 4: Advanced Types and Implicits

Type Parameters (Generics)

- Generics are a common feature in many modern statically typed languages.
- Generics allow the same code to be reused for multiple different types.
- Generics are an important tool for safe use of DRY.

Generics in Scala

- Scala has had generics for a long time, before Java did.

- Scala's generics are implemented via "type constructors", in which a type is created by passing other types to a generic type as arguments.

- Under the hood, generics are erased, which leads to certain important nuances.

A Generic Example

```scala
// a class with a type constructor
case class AnClass[A](x: A, f: A => A) {
    def mash(): A = f(x)
}

// you can construct the type
// and save is as a type variable...
type AnIntClass = AnClass[Int]

// and instantiate it
val anThing = new AnIntClass(1, _ + 2)

// or instantiate it immediately
val anOtherThing = new AnClass[String]("foo", _.toUpperCase)
```

Polymorphic Functions

- Like types, functions can have type parameters in addition to their value parameters.
- Also like types, the type parameters are between the name and the value parameter lists:

```
def foo[A](x: A): Int
```

Type Bounds

- Type parameters can be bound above and below by their inheritance relatives.

```scala
// bound above
class BoundAbove[A <: AnyRef](x: A)

// bound below
class BoundBelow[A >: String](x: A)

// both bounds
class BoundBoth[A <: Product >: Null](x: A)
```

Type Variance

```scala
class T; class S extends T

// Invariant[S] <: Invariant[T] is not true despite S <: T
class Invariant[A]

// Covariant[S] <: Covariant[T] is true because S <: T
class Covriant[+A]

// Contravariant[S] >: Contravariant[T] is true because S <: T
class Contravariant[-A]
```

Wildcards and `forSome`

- When Java 5 added generics, it included type wildcards, e.g. `ArrayList<?>`
- Scala's equivalent is forSome: `ArrayList[T forSome {type T}]`, or the shorthand `ArrayList[_]`
- `forSome` is also used for bounded wildcards.

`implicit`

- One of Scala's most powerful and abusable features is implicits, which allow the compiler to fill in code for you.

- There are two main types of implicits: implicit parameters and implicit conversions.

- Implicits are dangerous because misuse can make programs difficult to understand, and confuse the compiler.

Implicit Parameters

- Implicit parameters allow the compiler to take an unambiguous implicit val and insert it into the last parameter list of a function, if the list is marked `implicit`.

```
implicit val anImplicit: Int = 12

def iHazImplicit(i: Int)(implicit j: Int): Int = i + j

iHazImplicit(8) // 20
```

Implicit Conversions

- An implicit conversion is a single-argument function marked implicit, that will be used to automatically cast an object to another when a method for the other type is called on it.

```
// a conversion
// note that we can't use String.toInt
// because that invokes another ambiguous implicit
implicit def string2int(s: String): Int = Integer.parseInt(s)

val anInt = "2" - 2 // 0

// String already has the method +(x: Any): String
val anotherInt = "2" + 2 // "22"
```

Extension Methods

- Implicit classes can be used to add methods to existing classes, which can be used to turn function applications into simpler method calls.

```
// implicit class, extension method pattern
implicit class _double(val d: Double) extends AnyVal {
    def **(other: Double): Double = Math.pow(d, other)
}

val power = 2 ** 3 // 8.0
```

Context and View Bounds

- Context bounds are a shorthand for implicit parameters whose types have single type parameters.
- View bounds are like upper type bounds, except it requires there to exist some way to convert the type into the view bound.

```scala
def contextBound[A : ClassTag](x: A)
// is shorthand for
def contextBound[A](x: A)(implicit tag: ClassTag[A])

// a view bound
def asString[A <% String](x: A): String = x
```

Implicit Evidence

- Sometimes, functions may require further evidence to be inserted by the compiler, namely `=:=` and `<:<`.

- Having `A =:= B` as an implicit parameter requires the types `A` and `B` to be the same.

- Having `A <:< B` as an implicit parameter requires the types `A` to be a subtype of `B`.

Structural types

- Instead of requiring a type to implement a trait, you can simply require they have very specific method with structural types.
- Beware, structural types use reflection, and thus have a performance penalty.

```
def toUppercase(x: { def toUpperCase(): String }): String =
x.toUpperCase

toUpperCase("anString") // "ANSTRING"
```

Higher Kinds

- There are several issues with generics in Java, such as that `public interface Foo<A<?>>` is not valid; you can't require a type constructor as a type parameter without jumping though hoops.
- Scala allows this, with `class Foo[A[_]]`, which allows you to abstract over `A`'s parameter.
- Essentially, this is the type analogue to higher-order functions, or functions that take other functions as arguments.

Higher Kinds

- Higher kinds can be used to remove code repetition in a similar way that higher order functions do.

```
import language.higherKinds

trait AnTrait[A, B[_], ~>[_, _]] {
    def apply(x: A): A ~> B[A]
}

object AnObject extends AnTrait[Int, List, Function1] {
    def apply(x: Int): Int => List[Int] = _ :: x :: Nil
}
```

Abstract Type Members

- Like `val`s and `def`s, it is possible to make `type`s abstract.
- Concrete subtypes can fill in the abstract type members, as long as they match the bounds (including higher kindedness).