Scala = object-oriented + functional (5-minute lightning talk)

Franklin Chen
http://franklinchen.com/

October 14, 2014

Language features

Pure OO to the extreme

- *Everything* is an **object**!

  ```
  1 + 2 //... is syntactic sugar for:
  1.+(2)
  ```

- `class`: as in Java
- `trait`: like Java `interface`
  - but can contain code
  - multiple `trait`s can be mixed together

    ```
    class FancyService extends Service
      with Logging
      with Timeout
      with Authentication
    ```

Typed functional programming: basic

- (Underneath, functions are just objects)
- Type inference: puts in types before compilation happens

```
val s = "foobar".substring(1, 3) //... becomes:
val s: String = "foobar".substring(1, 3)
```

- Generic types

```
def zip[A, B](xs: List[A], ys: List[B]):
  List[(A, B)] = //...
```

- Higher-order functions

```
List(1, 2, 3).map(i => i+1)
```

Typed function programming: case classes

- Goodbye to Java getter/setter/constructor boilerplate!

  ```
  case class User(name: String, age: Int)

  val boy = User("jack", 2)
  println(boy.name)
  ```

- Pattern matching on case classes

  ```
  message match {
    case CheckIn(time, User(theName, theAge)) =>
      // ...
    case CheckOut(time, userId) =>
      // ...
  }
  ```

Typed functional programming (advanced)

Very powerful, tremendously useful:

- Implicits: Scala's main contribution to types
  - Compile-time monkey patching

    ```
    implicit class StringStuff(s: String) {
      def isSilly() = s.contains("silly")
    }
    val status = "somesillything".isSilly()
    ```

  - Type class pattern (more powerful than Haskell's)
  - Default contexts
- Structural types
  - Compile-time duck typing

    ```
    def makeNoise(duck: { def quack(): String }) =
      duck.quack() + "!!!"
    class FrenchDuck { def quack() = "coin" }
    val noise = makeNoise(new FrenchDuck())
    ```

- Higher-kinded types
- Existential types

Selling point: compatibility with Java

Enables *gradual transition* from Java without pain!

- Call Java easily from Scala

```
import java.util.Calendar
val time = Calendar.getInstance().getTime()
```

- Java can call Scala also
- JVM languages in general (Clojure, JRuby, etc.)

Pleasant, powerful syntax

- Don't need semicolons
- String interpolation

```
println(s"Name: ${user.name}; next age: ${user.age + 1}")
```

- Macros
  - Transform code before it reaches compiler
  - Example of code manipulation using *quasiquotes* (Lisp-inspired):

    ```
    scala> val scalaTree = q"foo(x + y)"
    scalaTree: universe.Tree = foo(x.$plus(y))

    scala> scalaTree match {
      case q"foo(x + $second)" => println(second)
    }
    y
    ```

  - Many libraries use macros to remove boilerplate, optimize, enable nice syntax

Ecosystem features

Build tool

- SBT: build tool, compiler as a service
  - Typesafe Activator: GUI and user-contributed project templates
- REPL
- Incremental compilation
- Incremental testing: rerun only tests affected by changed code

IDEs

- Scala IDE for Eclipse
- IntelliJ IDEA
- ENSIME for Emacs, etc.

Testing frameworks

- Example-based
  - ScalaTest
  - Specs2

    ```
    "Hello world" must endWith("world")
    ```

- Property-based
  - ScalaCheck

    ```
    forAll { (a: String, b: String) =>
      (a+b).startsWith(a)
    }
    ```

- doctest
  - Extracts tests embedded in comments in source code, runs tests

Many awesome libraries

- Akka: actor framework for concurrent, distributed programming
- Play: scalable Web framework built on Akka
- Spark: fast, elegant Big Data processing
  - Minimal boilerplate thanks to functional programming

    ```
    val wordCounts = textFile.
      flatMap(line => line.split(" ")).
      map(word => (word, 1)).
      reduceByKey((a, b) => a + b)
    ```

  - Beats Hadoop MapReduce in performance
- Scala Blitz: fast parallel collections
- Scala.js: compile to JavaScript
- Many others: awesome-scala