Welcome to Scala

Team Deck the Halls

Aaron Bamberger

Mitchell Garvin

Samad Jawaid

Scala is…

A. Object-Oriented?
B. Functional?
C. Imperative?
D. All of the Above!

A Brief History of Scala

Scala was created in 2001 at the École Polytechnique Fédérale de Lausanne, one of the Swiss Federal Institutes of Technology, by Martin Odersky, who had previously worked on the Sun Java compiler, javac, and on Java's generics system

Scala stands for scalable language.

Scala was designed with the philosophy that it should be easy to create small, targeted, application specific languages using Scala. To this end, Scala supports several ways of adding syntax-like features to the language without actually changing the syntax

Functions as operators

One feature that makes language extensions possible, is that all functions can by default be used as infix and postfix operators.

```
def myAdd(lArg: Int, rArg: Int) : Int =
    lArg + rArg


3 myAdd 4
```

Returns 7

Variable Definitions and Types

There are two types of variable definitions in scala:

`var` declares a mutable variable

`val` declares an immutable constant (like `const` in Java)

Scala is statically typed. Types follow variable declarations after a colon. For example:

```
val newString1: String = “Hi” //Immutable String

var newInt2: Int = 42 //Mutable Int
```

Objects vs. Classes

In most Object Oriented languages, it's difficult and verbose to create singleton objects. Scala makes it easy with it's distinction between Objects and Classes.

The `class` keyword declares classes as in Java or C++

The `object` keyword declares singleton objects, otherwise the semantics are the same as for classes

Objects vs. Classes cont.

```
//Class, can instantiate multiple instances
class myFoo1(arg1: int, arg2: String) =
{
    …
}

//Object, language enforced singleton instance
object myFoo2 =
{
    …
}
```

Function Definitions

Function definitions start with the `def` keyword. Function definitions are very similar to those in Java or C, but the type of the arguments and the return type are specified after a colon like variables.

```
def myFun(arg1: String, arg2: String): String =
{
    return arg1 + arg2
}
```

The special return type `Unit` functions like `void` in java. Use is as a return type for functions that don't return a value

```
def mySub(name: String): Unit =
{
    println("Hello, " + name + "!")
}
```

Hello, world! (Option a)

- Run `scala`
- Run `println("Hello, world!")`

Hello, world! (Option b)

- Create HelloWorld.scala

```
object HelloWorld {
    def main(args: Array[String]) {
        println("Hello, world!")
    }
}
```

- Run `scalac HelloWorld.scala`
  - This compiles the code.
- Run `scala HelloWorld`

Actors

Scala uses Actors for concurrency, in a method similar to Erlang. Actors are concurrent operations that execute asynchronously, and pass messages back and forth to communicate.

An actor is created by making a new class that inherits from Actor. The act() method in this class is overridden to provide the Actor's functionality.

The ! operator is used to send a message to an actor. Messages sent can be any value, but are usually instances of case classes

Actors Example

```
case object Message1

case object Message2


object main{

        def main(args: Array[String]) {

             newActor: Sender = new
        Sender()

             newActor.start

             for(i <- 0 until 10) {

                  if(i % 2  == 0)

                       newActor ! Message1

                  else

                       newActor ! Message2

             }

        }

}
```

```
class Sender() extends Actor{
          def act(){
                    while(true)
                    {
                              receive
                              {

          case Message1 =>

          println("Message1")

          case Message2 =>

          println("Message2")
                              }
                    }
          }
}
```