Basics

Next»

This lesson covers:

About this class

The first few weeks will cover basic syntax and concepts, then we'll start to open it up with more exercises.

Some examples will be given as if written in the interpreter and others as if written in a source file.

Having an interpreter available makes it easy to explore a problem space.

Why Scala?

- Expressive
  - First-class functions
  - Closures
- Concise
  - Type inference
  - Literal syntax for function creation
- Java interoperability
  - Can reuse java libraries
  - Can reuse java tools
  - No performance penalty

How Scala?

- Compiles to java bytecode
- Works with any standard JVM
  - Or even some non-standard JVMs like Dalvik
  - Scala compiler written by author of Java compiler

Think Scala

Scala is not just a nicer Java. You should learn it with a fresh mind- you will get more out of these classes.

Get Scala

Scala School's examples work with Scala 2.9.x . If you use Scala 2.10.x or newer, *most* examples work OK, but not all.

Start the Interpreter

Start the included `sbt console`.

```
$ sbt console

[...]

Welcome to Scala version 2.8.0.final (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_20).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Expressions

```
scala> 1 + 1
res0: Int = 2
```

res0 is an automatically created value name given by the interpreter to the result of your expression. It has the type Int and contains the Integer 2.

(Almost) everything in Scala is an expression.

Values

You can give the result of an expression a name.

```
scala> val two = 1 + 1
two: Int = 2
```

You cannot change the binding to a val.

Variables

If you need to change the binding, you can use a `var` instead.

```
scala> var name = "steve"
name: java.lang.String = steve

scala> name = "marius"
name: java.lang.String = marius
```

Functions

You can create functions with def.

```
scala> def addOne(m: Int): Int = m + 1
addOne: (m: Int)Int
```

In Scala, you need to specify the type signature for function parameters. The interpreter happily repeats the type signature back to you.

```
scala> val three = addOne(2)
three: Int = 3
```

You can leave off parens on functions with no arguments.

```
scala> def three() = 1 + 2
three: ()Int

scala> three()
res2: Int = 3

scala> three
res3: Int = 3
```

Anonymous Functions

You can create anonymous functions.

```
scala> (x: Int) => x + 1
res2: (Int) => Int = <function1>
```

This function adds 1 to an Int named x.

```
scala> res2(1)
res3: Int = 2
```

You can pass anonymous functions around or save them into vals.

```
scala> val addOne = (x: Int) => x + 1
addOne: (Int) => Int = <function1>
```

```
scala> addOne(1)
res4: Int = 2
```

If your function is made up of many expressions, you can use {} to give yourself some breathing room.

```
def timesTwo(i: Int): Int = {
  println("hello world")
  i * 2
}
```

This is also true of an anonymous function.

```
scala> { i: Int =>
  println("hello world")
  i * 2
}
res0: (Int) => Int = <function1>
```

You will see this syntax often used when passing an anonymous function as an argument.

Partial application

You can partially apply a function with an underscore, which gives you another function. Scala uses the underscore to mean different things in different contexts, but you can usually think of it as an unnamed magical wildcard. In the context of `{ _ + 2 }` it means an unnamed parameter. You can use it like so:

```
scala> def adder(m: Int, n: Int) = m + n
adder: (m: Int,n: Int)Int
```

```
scala> val add2 = adder(2, _:Int)
add2: (Int) => Int = <function1>

scala> add2(3)
res50: Int = 5
```

You can partially apply any argument in the argument list, not just the last one.

Curried functions

Sometimes it makes sense to let people apply some arguments to your function now and others later.

Here's an example of a function that lets you build multipliers of two numbers together. At one call site, you'll decide which is the multiplier and at a later call site, you'll choose a multiplicand.

```
scala> def multiply(m: Int)(n: Int): Int = m * n
multiply: (m: Int)(n: Int)Int
```

You can call it directly with both arguments.

```
scala> multiply(2)(3)
res0: Int = 6
```

You can fill in the first parameter and partially apply the second.

```
scala> val timesTwo = multiply(2) _
timesTwo: (Int) => Int = <function1>

scala> timesTwo(3)
res1: Int = 6
```

You can take any function of multiple arguments and curry it. Let's try with our earlier `adder`

```
scala> val curriedAdd = (adder _).curried
curriedAdd: Int => (Int => Int) = <function1>

scala> val addTwo = curriedAdd(2)
```

```
addTwo: Int => Int = <function1>

scala> addTwo(4)
res22: Int = 6
```

Variable length arguments

There is a special syntax for methods that can take parameters of a repeated type. To apply String’s `capitalize` function to several strings, you might write:

```
def capitalizeAll(args: String*) = {
  args.map { arg =>
    arg.capitalize
  }
}

scala> capitalizeAll("rarity", "applejack")
res2: Seq[String] = ArrayBuffer(Rarity, Applejack)
```

Classes

```
scala> class Calculator {
     |   val brand: String = "HP"
     |   def add(m: Int, n: Int): Int = m + n
     | }
defined class Calculator

scala> val calc = new Calculator
calc: Calculator = Calculator@e75a11

scala> calc.add(1, 2)
res1: Int = 3

scala> calc.brand
res2: String = "HP"
```

Contained are examples defining methods with def and fields with val. Methods are just functions that can access the state of the class.

Constructor

Constructors aren’t special methods, they are the code outside of method definitions in your class. Let’s extend our Calculator example to take a constructor argument and use it to initialize internal state.

```
class Calculator(brand: String) {
  /**
   * A constructor.
   */
  val color: String = if (brand == "TI") {
    "blue"
  } else if (brand == "HP") {
    "black"
  } else {
    "white"
  }

  // An instance method.
  def add(m: Int, n: Int): Int = m + n
}
```

Note the two different styles of comments.

You can use the constructor to construct an instance:

```
scala> val calc = new Calculator("HP")
calc: Calculator = Calculator@1e64cc4d

scala> calc.color
res0: String = black
```

Expressions

Our Calculator example gave an example of how Scala is expression-oriented. The value color was bound based on an if/else expression. Scala is highly expression-oriented: most things are expressions rather than statements.

Aside: Functions vs Methods

Functions and methods are largely interchangeable. Because functions and methods are so similar, you might not remember whether that *thing* you call is a function or a method. When you bump into a difference between methods and functions, it might confuse you.

```
scala> class C {
     |   var acc = 0
     |   def minc = { acc += 1 }
     |   val finc = { () => acc += 1 }
     | }
defined class C

scala> val c = new C
c: C = C@1af1bd6

scala> c.minc // calls c.minc()

scala> c.finc // returns the function as a value:
res2: () => Unit = <function0>
```

When you can call one "function" without parentheses but not another, you might think *Whoops, I thought I knew how Scala functions worked, but I guess not. Maybe they sometimes need parentheses?* You might understand functions, but be using a method.

In practice, you can do great things in Scala while remaining hazy on the difference between methods and functions. If you're new to Scala and read explanations of the differences, you might have trouble following them. That doesn't mean you're going to have trouble using Scala. It just means that the difference between functions and methods is subtle enough such that explanations tend to dig into deep parts of the language.

Inheritance

```
class ScientificCalculator(brand: String) extends Calculator(brand) {
  def log(m: Double, base: Double) = math.log(m) / math.log(base)
}
```

See Also Effective Scala points out that a Type alias is better than `extends` if the subclass isn't actually different from the superclass. A Tour of Scala describes Subclassing.

Overloading methods

```
class EvenMoreScientificCalculator(brand: String) extends ScientificCalculator(brand) {
  def log(m: Int): Double = log(m, math.exp(1))
}
```

Abstract Classes

You can define an *abstract class*, a class that defines some methods but does not implement them. Instead, subclasses that extend the abstract class define these methods. You can't create an instance of an abstract class.

```
scala> abstract class Shape {
     |   def getArea():Int    // subclass should define this
     | }
defined class Shape

scala> class Circle(r: Int) extends Shape {
     |   def getArea():Int = { r * r * 3 }
     | }
defined class Circle

scala> val s = new Shape
<console>:8: error: class Shape is abstract; cannot be instantiated
       val s = new Shape
               ^

scala> val c = new Circle(2)
c: Circle = Circle@65c0035b
```

Traits

`traits` are collections of fields and behaviors that you can extend or mixin to your classes.

```
trait Car {
  val brand: String
}

trait Shiny {
  val shineRefraction: Int
}
```

```
class BMW extends Car {
  val brand = "BMW"
}
```

One class can extend several traits using the `with` keyword:

```
class BMW extends Car with Shiny {
  val brand = "BMW"
  val shineRefraction = 12
}
```

See Also Effective Scala has opinions about trait.

When do you want a Trait instead of an Abstract Class? If you want to define an interface-like type, you might find it difficult to choose between a trait or an abstract class. Either one lets you define a type with some behavior, asking extenders to define some other behavior. Some rules of thumb:

- Favor using traits. It's handy that a class can extend several traits; a class can extend only one class.
- If you need a constructor parameter, use an abstract class. Abstract class constructors can take parameters; trait constructors can't. For example, you can't say `trait t(i: Int) {}`; the `i` parameter is illegal.

You are not the first person to ask this question. See fuller answers at stackoverflow:Scala traits vs abstract classes, Difference between Abstract Class and Trait, and Programming in Scala: To trait, or not to trait?

Types

Earlier, you saw that we defined a function that took an `Int` which is a type of Number. Functions can also be generic and work on any type. When that occurs, you'll see a type parameter introduced with the square bracket syntax. Here's an example of a Cache of generic Keys and Values.

```
trait Cache[K, V] {
  def get(key: K): V
  def put(key: K, value: V)
  def delete(key: K)
}
```

Methods can also have type parameters introduced.

```
def remove[K](key: K)
```