CS 2340 Objects and Design - Scala

Objects and Operators

Christopher Simpkins
chris.simpkins@gatech.edu

Classes

- A class is a blueprint for creating (or instantiating) objects
- An object is an instance of a class, created using `new`
- In Scala, every entity you can refer to is an object (or type - more later)

Person Example

```
class Person {

  val firstName: String = ``John''
  val lastName: String = ``Doe''
  var sex: String = "male"

  override def toString = {
    firstName + " " + lastName
  }
}
```

Instance Variables

- Represent the state of the object
- Public by default, can specify `private` or `protected` - more later
- `val` instance variables are immutable
- `var` instance variables are mutable
- Must be initialized

Instance Variables Examples

`var`s can be reassigned:

```
val mrGarrison = new Person
mrGarrison.sex = ``male''
mrGarrison.sex = ``female''
```

`val`s cannot be reassigned:

```
val jackDoe = new Person
jackDoe.firstName = ``Jack'' // won't compile - reassignment to val
```

Primary Constructor

```
class Person(val firstName: String,
             val lastName: String,
             var sex: String = "male") {

  override def toString = {
    firstName + " " + lastName
  }
}
```

Now persons can be created like this:

```
val mrGarrison = new Person("Mr.", "Garrison")
val grace = new Person("Grace", "Hopper", "female")
```

Methods

- Definition starts with `def` keyword
- Every method has a type
- Some methods don't have a type. Their type is `Unit`
- `return` statement is optional and discouraged - last value evaluated in a method is return value

Method Example

```
class Foo {

  def nCopies(s: String, n: Int): String = {
    s*n
  }

  def printNCopies(s: String, n: Int) { // Notice no ``=''
    print(nCopies(s, n))
  }
}
```

Applications

- An `object` (singleton) with a `main` method is runnable

```
object App {

  def main(args: Array[String]) {
    println(``Arguments are:'')
    for (arg <- args) {
      println(arg)
    }
  }
}
```

Basic Types and Operations

- Integral types: `Byte`, `Short`, `Int`, `Long`, and `Char`
- Numeric types: integral types plus `Float` and `Double`
- Scala's `String` uses `java.lang.String` directly (for now)
- Numeric types are *value types*, which means that they are implemented directly as Java primitives in bytecode but are decorated by "rich wrappers" in Scala which allow you to call methods on basic types.

Operators are Methods

```
scala> 1 + 2
res2: Int = 3

scala> 1.+(2)
res3: Double = 3.0

scala> (1).+(2)
res4: Int = 3
```

Unary operators can be defined for +, -, !, and ~. The method name is, e.g., `unary_!` for unary `!`.

```
scala> !true
res3: Boolean = false

scala> true.unary_!
res4: Boolean = false
```

Basic Operations

- Precedence, bitwise operators like Java - read on your own
- Scala twist: any method that ends in a `:` character is invoked on its right operand, passing in the left operand.

Example: `::` (pronounced “cons”) is right-associative. It is called on the `List` that is its right operand.

```
scala> val ys = List(2, 3)
ys: List[Int] = List(2, 3)

scala> val zs = 1::ys
zs: List[Int] = List(1, 2, 3)
```

Note that operands are always evaluated left to right:

```
scala> val as = List(4, 5)
as: List[Int] = List(4, 5)

scala> 1+2::as
res7: List[Int] = List(3, 4, 5)
```

Object Equality

== and `equals` test *value* equality, `eq` tests reference equality

```
scala> val xs = List(1, 2, 3)
xs: List[Int] = List(1, 2, 3)

scala> val ys = xs
ys: List[Int] = List(1, 2, 3)

scala> val zs = List(1, 2, 3)
zs: List[Int] = List(1, 2, 3)

scala> xs == ys
res8: Boolean = true

scala> xs == zs
res9: Boolean = true

scala> xs eq ys
res10: Boolean = true

scala> xs eq zs
res11: Boolean = false
```