TOUR OF SCALA

SINGLETON OBJECTS

An object is a class that has exactly one instance. It is created lazily when it is referenced, like a lazy val.

As a top-level value, an object is a singleton.

As a member of an enclosing class or as a local value, it behaves exactly like a lazy val.

Defining a singleton object

An object is a value. The definition of an object looks like a class, but uses the keyword `object`:

```
object Box
```

Here's an example of an object with a method:

```
package logging

object Logger {
  def info(message: String): Unit = println(s"INFO: $message")
}
```

The method `info` can be imported from anywhere in the program. Creating utility methods like this is a common use case for singleton objects.

Let's see how to use `info` in another package:

```
import logging.Logger.info

class Project(name: String, daysToComplete: Int)

class Test {
  val project1 = new Project("TPS Reports", 1)
  val project2 = new Project("Website redesign", 5)
  info("Created projects")  // Prints "INFO: Created projects"
}
```

The `info` method is visible because of the import statement, `import logging.Logger.info`.

Imports require a "stable path" to the imported symbol, and an object is a stable path.

Note: If an `object` is not top-level but is nested in another class or object, then the object is "path-dependent" like any other member. This means that given two kinds of beverages, `class Milk` and `class OrangeJuice`, a class member `object NutritionInfo` "depends" on the enclosing instance, either milk or orange juice. `milk.NutritionInfo` is entirely distinct from `oj.NutritionInfo`.

Companion objects

An object with the same name as a class is called a *companion object*. Conversely, the class is the object's companion class. A companion class or object can access the private members of its companion. Use a companion object for methods and values which are not specific to instances of the companion class.

```
import scala.math._

case class Circle(radius: Double) {
  import Circle._
  def area: Double = calculateArea(radius)
}

object Circle {
  private def calculateArea(radius: Double): Double = Pi * pow(radius, 2.0)
}

val circle1 = Circle(5.0)

circle1.area
```

The `class Circle` has a member `area` which is specific to each instance, and the singleton `object Circle` has a method `calculateArea` which is available to every instance.

The companion object can also contain factory methods:

```
class Email(val username: String, val domainName: String)

object Email {
  def fromString(emailString: String): Option[Email] = {
    emailString.split('@') match {
      case Array(a, b) => Some(new Email(a, b))
      case _ => None
    }
  }
}

val scalaCenterEmail = Email.fromString("scala.center@epfl.ch")
scalaCenterEmail match {
  case Some(email) => println(
    s"""Registered an email
       |Username: ${email.username}
       |Domain name: ${email.domainName}
     """.stripMargin)
  case None => println("Error: could not parse email")
}
```

The `object Email` contains a factory `fromString` which creates an `Email` instance from a String. We return it as an `Option[Email]` in case of parsing errors.

Note: If a class or object has a companion, both must be defined in the same file. To define companions in the REPL, either define them on the same line or enter `:paste` mode.

Notes for Java programmers

`static` members in Java are modeled as ordinary members of a companion object in Scala.

When using a companion object from Java code, the members will be defined in a companion class with a `static` modifier. This is called *static forwarding*. It occurs even if you haven't defined a companion class yourself.

More resources

- Learn more about Companion objects in the Scala Book

← previous next →

Contributors to this page:

ckipp01 mlachkar erikvanzijst Zukkari ashawley som-snytt travissarles heathermiller