First Objects

Otfried Cheong

February 18, 2012

1 Classes and objects

Objects are the basis of object-oriented programming. In Scala, every piece of data is an object. Each object has a *type*, such as `Int`, `Double`, `String`, tuple, or `Array[Int]`. The type of an object determines what you can do with the object. When you use an object, you should think of the object as a black box, and you need not know what really happens inside the object. All you need to know is what the object does, not how it implements its functionality.

A class defines a *new type* of object. You can also think about a class as a "blueprint" for objects. Once you define a class, you can create objects from the blueprint.

2 Case classes

A common use of classes is to define objects with a number of attributes. For instance:

- A point in the plane has an x-coordinate and a y-coordinate. Depending on the application, the coordinates could be integers, or floating-point numbers.
- A date has a year, month, and day.
- A student object has (at least) a name, a student number, and a major.
- A playing card (such as in Blackjack) has a suit (clubs, spades, hearts, or diamonds) and a face value (2,3,...,10, Jack, Queen, King, Ace).

In Scala, such a simple class is best implemented as a *case class*:

```
scala> case class Point(x: Int, y: Int)
defined class Point
```

`Point` represents a two-dimensional point. It has two *fields*, namely `x` and `y`. We can create `Point` objects as follows:

```
scala> var p = Point(2, 5)
p: Point = Point(2,5)
```

This looks like a function call, and in fact it is a call to the *constructor* for the `Point` class.

Once we have a `Point` object, we can access its fields using *dot*-syntax:

```
scala> p.x
res0: Int = 2
scala> p.y
res1: Int = 5
```

Note that we *cannot modify* the fields of the `Point` object:

```
scala> p.x = 7
<console>:10: error: reassignment to val
```

In other words, once a `Point` object has been created, its fields cannot be modified. We say that `Point` is *immutable* (unchangeable).

We can compare two `Point` objects using `==` and `!=`. Two case classes are equal if and only if all their fields are equal.

```
scala> val q = Point(7, 19)
q: Point = Point(7,19)
scala> val r = Point(2, 5)
r: Point = Point(2,5)
scala> p == r
res2: Boolean = true
scala> p == q
res3: Boolean = false
scala> p != q
res4: Boolean = true
```

Let's now look at the other examples from above: A date object could be defined like this:

```
scala> case class Date(year: Int,
                       month: Int,
                       day: Int)
defined class Date
scala> val d = Date(2012, 2, 20)
d: Date = Date(2012,2,20)
scala> d.month
res5: Int = 2
scala> d.day
res6: Int = 20
```

A student object might look like this:

```
scala> case class Student(name: String,
                  id: Int, dept: String)
defined class Student
scala> val s = Student("Otfried",13,"CS")
s: Student = Student(Otfried,13,CS)
scala> s.id
res7: Int = 13
```

And a Blackjack card object could look like this:

```
scala> case class Card(face: String,
                       suit: String)
defined class Card
scala> val c = Card("Ace", "Diamonds")
c: Card = Card(Ace,Diamonds)
scala> c.suit
res8: String = Diamonds
```

3 Immutable and mutable objects

An object whose state cannot change after it has been constructed is called *immutable*. The methods of an immutable object do not modify the state of the object. In Scala, all number types, strings, and tuples are immutable. The classes `Point`, `Date`, `Student`, and `Card` we defined above are all immutable.

If we want to define a *mutable* case class, we need to put the `var` keyword in front of the field names:

```
scala> case class Point2(var x: Int,
                         var y: Int)
defined class Point2
scala> val p = Point2(3, 5)
p: Point2 = Point2(3,5)
scala> p.x = 7
p.x: Int = 7
scala> p
res9: Point2 = Point2(7,5)
```

Mutable objects can lead to tricky mistakes. Consider the following code:

```
scala> val p = Point2(3, 5)
p: Point2 = Point2(3,5)
scala> val q = p
q: Point2 = Point2(3,5)
scala> q.x = 7
q.x: Int = 7
scala> q
res10: Point2 = Point2(7,5)
```

What is the value of `p` at this point? Surprisingly, `p` has changed as well:

```
scala> p
res11: Point2 = Point2(7,5)
```

Arrays are of course mutable, and so the same effect can appear for arrays:

```
scala> val A = Array(1, 2, 3, 4)
A: Array[Int] = Array(1, 2, 3, 4)
scala> val B = A
B: Array[Int] = Array(1, 2, 3, 4)
scala> A(2) = 99
scala> B
res1: Array[Int] = Array(1, 2, 99, 4)
```

Note that even though we have defined `A` as a `val` variable, it is possible to change the contents of `A`. The word `val` here only means that the meaning of the name `A` will never change—`A` is always the same array object. But what is *in* the array with name `A` can change.

4 References and the heap

Why does this happen? To understand this, we need to understand how variables store objects.

All Scala objects are stored in an area of the Scala runtime system called the *heap*. Objects cannot exist anywhere else.

A *variable* is just a *name* for an object on the heap. You can think about a variable as a *reference* to the object on the heap. The reference uniquely indicates the object on the heap. (If you learnt C, you can think about this reference as a pointer. In reality it may not really be a memory address.)

An assignment operation (as in `val q = p` or `val B = A` above) only copies the *reference*. Afterwards, `p` and `q` contain a reference to the *same* `Point2` object on the heap, and `B` and `A` contain a reference to the same array object.

So what happened in the examples above is that we created a *new name* for an object on the heap. `p` and `q` are in fact two different names for the *same* `Point2` object, and `A` and `B` are two names for the *same* array object.

This problem can never happen for *immutable* objects, and so it is preferable to use immutable objects whenever that is possible.

5 null

A variable can also have the value `null`, which means that it does *not* reference any object. For efficiency reasons, variables of the types `Int`, `Byte`, `Short`, `Long`, `Double`, `Float`, `Char`, `Boolean`, and `Unit` *cannot* be `null`.

6 Local variables

The local variables of a method are stored inside the method's *activation record* (also called *stack frame*). The activation record is created each time the method is called. This method

```
def test(m: Int) {
  val k = m + 27
  val s = "Hello World"
  val A = Array( s.length, k, m )
}
```

has four local variables, namely `m`, `k`, `s`, and `A`. (The parameters of a method are local `val` variables, with the only difference that the runtime system automatically places a value in the variable when the method is called.)

In this example, if `test(13)` is called, an activation record with space for the four variables is created.

7 Garbage collection

Scala objects are garbage collected: If the runtime system runs out of memory, it will check all the objects on the heap. If an object no longer has any reference pointing to it, the object is no longer useful, and will be deleted. It is hard to predict when garbage-collection will happen. If you run a small program only, probably no garbage-collection at all occurs.

Garbage collection allows the programmer not to worry about the memory management. There are other languages which do not provide an automatic garbage collection. For example, in C++ the programmer is responsible for the memory management. It is common for C or C++ programs to contain mistakes where objects are created but never destroyed, and so more and more unused and unusable objects fill up the heap. Such a program is said to contain a *memory leak*.

8 Arrays

Arrays are somewhat special objects—they are the only object that allows you to store an unbounded amount of information in a single object. (Scala provides many other classes to store large amounts of data efficiently and more conveniently than with arrays. But all of those classes are implemented internally using arrays or a large number of small objects.)

Arrays have a fixed size, and so we have to know the size of the array when we create the array object. If we want to put more objects in an array than the original array size allows, we need to create a new array and copy the data from the old array to the new one. The array methods `++` and `:+` do this for us. The code below shows how we could do it manually:

```
var A = new Array[Int](10)
// .... many computations ...
// now we need more space in A
val B = new Array[Int](20)
for (int i <- 0 until A.length)
  B(i) = A(i)
A = B
// old A will be garbage-collected
```

Note that to do this "trick", we had to declare `A` as a `var` variable, otherwise the assignment `A = B` would not have worked.

9 Using library classes

It is not necessary to declare classes that one wants to use (unlike in Python, C, or C++) if one has set up the library path correctly. The Scala compiler and run time system will look for the classes automatically.

Documentation for the Scala library is here: `http://www.scala-lang.org/api/current/`

Even though Scala is a new language, there are already many libraries we can use, since Scala can use Java libraries. There are thousands of classes in various Java libraries, organized into packages. You can find useful packages and reuse those. Documentation about the standard Java library classes can be found in the Java API reference: `http://download.oracle.com/javase/6/docs/api/`