Classes and objects in Scala

Otfried Cheong

April 17, 2012

1 Classes and objects

Objects are the basis of object-oriented programming. In Scala, every piece of data is an object. The type of the object determines what you can do with the object. Classes can contain data (the *state* of the object) and *methods* (*what* you can do with the object). You can think about a class as a blueprint for objects. Once you define a class, you can create objects from the blueprint using the keyword `new`.

2 Consistency

A *date* consists of three attributes: year, month, and day, all of them integers. We could thus store a date as a triple `(Int, Int, Int)`, or using a simple case class with three attributes.

Both methods, however, do not guarantee that our date objects will be *consistent*. Consistency means that the attributes take on only legal, meaningful values, and that the values of each attribute are consistent with each other. For instance, a day value of 31 is consistent with a month value of 3, but not with a month value of 4. A day value of 29 and a month value of 2 are consistent only if the year value indicates a leap year.

Here is a `Date` class that ensures that its state is always consistent:

```
val monthLength =
    Array(31, 29, 31, 30, 31, 30,
          31, 31, 30, 31, 30, 31)

class Date(val year: Int, val month: Int,
           val day: Int) {
  require(1901 <= year && year <= 2099)
  require(1 <= month && month <= 12)
  require(1 <= day &&
          day <= monthLength(month - 1))
  require(month != 2 || day <= 28 ||
          (year % 4) == 0)
}
```

The four `require` statements inside the class *body* are executed every time an object of type `Date` is constructed. If the values are okay, nothing happens. Otherwise `require` throws an exception, and we know immediately that something is wrong. Since the `Date` object is immutable, the consistent state that is guaranteed when the object is constructed can never be broken and made inconsistent.

It is helpful if objects guarantee that their state is consistent. It makes it possible for functions working with the objects to proceed without error checking, and simplifies debugging since we will notice quickly when something went wrong.

Here are some examples for using `Date` objects:

```
scala> var d1 = new Date(2012, 4, 16)
d1: Date = Date@1f6c439
scala> println(d1.year, d1.month, d1.day)
(2012,4,16)
scala> var d2 = new Date(2012, 2, 30)
IllegalArgumentException: requirement failed
```

3 Methods

In the past, we had functions that took arguments of a particular type, such as a function `date2num` that converted a date, represented as year, month, and day, into a day index starting on January 1, 1901.

In object-oriented programming, we prefer to define functions that work on a specific type as a method of that type. One advantage is that it clearly documents the available functions for a given type (for instance, to find out what you can do with a `String`, you would look at the list of methods of the `String` class). The main advantage, however, will be the possibility of hiding or protecting information inside the object, as we will see later.

For the moment, let us add methods to convert a `Date` object into a day index, and to return the day of the week of a `Date`:

```
val weekday =
  Array("Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday",
        "Sunday")

class Date(val year: Int, val month: Int,
           val day: Int) {
  require(1901 <= year && year <= 2099)
  require(1 <= month && month <= 12)
  require(1 <= day &&
          day <= monthLength(month - 1))
  require(month != 2 || day <= 28 ||
          (year % 4) == 0)

  // returns the number of days
  // since 1901/01/01 (day 0)
  def dayIndex: Int = {
    val fourY = (365 + 365 + 365 + 366)
    val yearn = year - 1901
    var total = 0
    total += fourY * (yearn / 4)
    total += 365 * (yearn % 4)
    for (m <- 0 until month - 1)
      total += monthLength(m)
    total += day - 1
    if (year % 4 != 0 && month > 2)
      total -= 1
    total
  }

  def dayOfWeek: String =
    weekday((dayIndex + 1) % 7)
}
```

A method looks like a function definition, but is placed inside the body of the class. The body of the method can refer to the names of the object's fields as if they were global variables. Of course they are not: To call a method, we need an object, and the method refers to the fields of *that* object:

```
scala> var d1 = new Date(2012, 4, 16)
d1: Date = Date@1c97c3e
scala> var d2 = new Date(2000, 1, 1)
d2: Date = Date@165de14
scala> d1.dayIndex
res1: Int = 40648
scala> d2.dayIndex
res2: Int = 36159
scala> d1.dayOfWeek
res3: String = Monday
scala> d2.dayOfWeek
res4: String = Saturday
```

The two methods we wrote here do not take any arguments, not even empty parentheses, and so calling them looks exactly like accessing a field of the object. Scala allows this "uniform access" to fields and methods—it would be impossible in Java or C++.

4 Printing pretty dates

Every object in Scala (or Java) can be converted to a string, and that is what happens when you look at an object in the interactive mode, or using `println`: the object is first converted to a string using its `toString` method:

```
scala> d1
res5: Date = Date@14096e6
scala> val s = d1.toString
s: String = Date@14096e6
```

By default, the result of this conversion is not pretty: It contains the name of the class and a hexadecimal number (a number in base 16) that identifies the object on the heap.

We can change this by *overriding* the default definition of the `toString` method, like this:

```
class Date(val year: Int, val month: Int,
           val day: Int) {
  // as before
  override def toString: String =
    "%s, %s %d, %d".format(dayOfWeek,
       monthname(month-1), day, year)
}
```

Note the keyword `override`. With this method, our object looks prettier:

```
scala> var d1 = new Date(2012, 4, 16)
d1: Date = Monday, April 16, 2012
scala> val s = d1.toString
s: String = Monday, April 16, 2012
scala> println(d1)
Monday, April 16, 2012
```

5 Operators are methods

In Scala, there is no difference between operators like `+` and methods like `take`—they are all implemented as methods.

Any method that has exactly one argument can be called using *operator syntax*. For instance, the `Array[Int]` class has a method `take(n)` to take the first `n` elements, and a method `:+(el)` to add an element at the end. (Both methods return a new array, of course, since arrays cannot change their length.)

Here are examples that show that we can call methods either in *method syntax* or in *operator syntax*:

```
scala> val A = Array(1, 2, 3, 4, 5)
A: Array[Int] = Array(1, 2, 3, 4, 5)
scala> A.take(3)
res0: Array[Int] = Array(1, 2, 3)
scala> A take 3
res1: Array[Int] = Array(1, 2, 3)
scala> A :+ 9
res2: Array[Int] = Array(1, 2, 3, 4, 5, 9)
scala> A.:+(9)
res3: Array[Int] = Array(1, 2, 3, 4, 5, 9)
```

Let's add a difference operator `-` to our `Date` class, to return the difference between two dates as a number of days:

```
class Date(...) {
  // ...
  def -(rhs: Date) =
    dayIndex - rhs.dayIndex
}
```

We can now calculate with dates:

```
scala> val birth = new Date(1993, 7, 9)
birth: Date = Friday, July 9, 1993
scala> val today = new Date(2012, 4, 16)
today: Date = Monday, April 16, 2012
scala> today - birth
res0: Int = 6856
```

It can be fun to define your own operators, but don't go overboard—it is only useful when it makes the code more readable!

6 Overloading

Scala, like Java and C++ but unlike C, allows the *overloading* of method names: It is allowed to have different methods that have the same name, and only differ in the type of arguments they accept. Here is a simple example:

```
def f(n: Int) {
  println("Int " + n)
}
def f(s: String) {
  println("String " + s)
}
f(17)
f("CS109")
```

The compiler correctly determines that `f(17)` is a call to the first function, while `f("CS109")` is a call to the second function. (This example will not work if you type it into the interactive mode—you have to run it as a script!)

We can make use of overloading to add more operators to our `Date` class. We will allow adding or subtracting a number of days to a date to obtain a new date:

```
class Date(...) {
  // ...
  def num2date(n: Int): Date = {
    val fourY = (365 + 365 + 365 + 366)
    var year = 1901 + (n / fourY) * 4
    var day = n % fourY
    if (day >= 365 + 365 + 365 + 59) {
      year += 3
      day -= 365 + 365 + 365
    } else {
      year += (day / 365)
      day = day % 365
      if (day >= 59)
        day += 1
    }
    var month = 1
    while (day >= monthLength(month-1)) {
      day -= monthLength(month-1)
      month += 1
    }
    new Date(year, month, day+1)
  }

  def -(rhs: Date) =
      dayIndex - rhs.dayIndex

  def +(n: Int): Date =
      num2date(dayIndex + n)
  def -(n: Int): Date =
      num2date(dayIndex - n)
```

Note that there are *two* `-` operators defined for the `Date` class. The compiler correctly selects the one we need depending on the type of the right-hand side:

```
scala> val birth = new Date(1992, 8, 21)
birth: Date = Friday, August 21, 1992
scala> val baekil = birth + 100
baekil: Date = Sunday, November 29, 1992
scala> val today = new Date(2012, 4, 16)
today: Date = Monday, April 16, 2012
scala> today - birth
res1: Int = 7178
scala> today - 7178
res2: Date = Friday, August 21, 1992
```

7 Class arguments

Class arguments are the arguments that appear in paratheses after the class name in the class declaration. Class arguments can be `val` fields, `var` fields, or simply parameters for the construction of the class.

Here is an immutable `Point` class. Both class parameters are `val` fields:

```
class Point(val x: Int, val y: Int) {
  override def toString: String =
    "(%d, %d)".format(x, y)
}
```

Here is a mutable `Rect` class to store an axis-parallel rectangle. The class parameters are `var` fields:

```
class Rect(var corner: Point,
           var width: Int,
           var height: Int) {
  require(width > 0 && height > 0)
  override def toString: String =
    "[%d ~ %d, %d ~ %d]".format(corner.x,
      corner.x + width, corner.y,
      corner.y + height)
}
```

Here is a different version of this `Rect` class, where the `corner` field is defined in the body of the class:

```
class Rect(x: Int, y: Int, var width: Int,
           var height: Int) {
  var corner = new Point(x, y)
  require(width > 0 && height > 0)
  override def toString: String =
    "[%d ~ %d, %d ~ %d]".format(corner.x,
      corner.x + width, corner.y,
      corner.y + height)
}
```

Every occurrence of `val` or `var` in the body of the class defines a field of the class. Every field *has to be initialized* with a starting value, which is used when the object is constructed.

8 Privacy

Let us define an `Accumulator`, a counter that starts with zero, and to which we can add a value:

```
class Accumulator {
  var sum = 0
  def add(n: Int) {
    sum += n
  }
}
```

The class itself has no class parameters this time, so no arguments are given when we create objects with `new`:

```
var acc1 = new Accumulator
acc1.add(7)
acc1.add(13)
println(acc1.sum)
```

9 Privacy

What is not so nice here is that we could accidentally modify the `sum`-field:

```
scala> val acc2 = new Accumulator
acc2: Accumulator = Accumulator@5e7663
scala> acc2.add(17)
scala> acc2.add(23)
scala> acc2.sum = 0  // ouch!
scala> acc2.add(19)
scala> acc2.sum
res7: Int = 19
```

The programmer using the `Accumulator` class made a mistake here and set `acc2.sum` back to zero—so now the final result is wrong.

This is an example that shows the importance of privacy. A *client*—that is, code using our class—should consider an object as a black box. The client should not need or want to know about how the object is implemented, and only use the methods provided by the object to communicate with it. Our `Accumulator` object should have two operations: adding a number to the current sum, and reading out the current sum. It should not be possible to modify the current sum.

We can achieve this by forbidding client code to access the field `sum`. To do so, we declare the field to be `private`. However, that means that we cannot access it at all, so we have to add a new method to be able to read the current value of the counter:

```
class Accumulator {
  private var sum = 0
  def add(n: Int) {
    sum += n
  }
  def value: Int = sum
}
```

Here is how we use it:

```
scala> val acc = new Accumulator
acc: Accumulator = Accumulator@113e371
scala> acc.add(19)
scala> acc.add(4)
scala> acc.sum = 0
<console>:7: error: variable sum cannot
             be accessed in Accumulator
scala> println(acc.sum)
<console>:8: error: variable sum cannot
             be accessed in Accumulator
scala> println(acc.value)
23
```

Note how Scala stops us from changing or even looking at the value of `acc.sum`.

The `private` keyword means that the member can be accessed only from methods inside the class. You can use it both for fields and for methods. So a private method is a method that can be called only from other methods in the same class.

10 Constructors

Often it is not enough to just set the fields to some initial value when you construct a new object. You can then write the necessary computations directly inside the class body. Everything that is not a method definition is executed when the object is constructed—but remember that when you declare a variable with `var` or `val`, you are creating a field of the object.

As an example, let us define a `Deck` class that stores a deck of blackjack cards. The cards are stored in an array field `cards`. When the `Deck` object is constructed, this array must be filled with all 52 cards.

```
val Suits = Array("Clubs", "Spades",
                  "Hearts", "Diamonds")
val Faces = Array("2", "3", "4", "5",
                  "6", "7", "8", "9",
                  "10", "Jack", "Queen",
                  "King", "Ace")
```

```
class Deck {
  private val cards = new Array[Card](52)
  generateDeck()

  private def generateDeck() {
    var i = 0
    for (suit <- Suits) {
      for (face <- Faces) {
        cards(i) = new Card(face, suit)
        i += 1
      }
    }
  }
}
```

Note that `generateDeck` is a private method—it is only needed when the object is constructed, and should not be called again by client code.

So far the deck doesn't let us do much. We want to be able to draw cards from the top of the deck:

```
class Deck {
  private val cards = new Array[Card](52)
  private var count = 52
  generateDeck()

  def draw(): Card = {
    assert(count > 0)
    count -= 1
    cards(count)
  }
  private def generateDeck() // not shown
}
```

We can use the `Deck` class like this:

```
val deck = new Deck
for (i <- 1 to 10)
  println(deck.draw())
```

When we run this program, we realize that we forgot to shuffle the deck. The following final version fixes this problem.

```
class Deck {
  private val cards = new Array[Card](52)
  private var count = 52
  generateDeck()
  shuffleDeck()

  private def generateDeck() {
    var i = 0
    for (suit <- Suits) {
      for (face <- Faces) {
        cards(i) = new Card(face, suit)
        i += 1
      }
    }
  }
  private def shuffleDeck() {
    for (i <- 1 to 52) {
      // 0..i-2 already shuffled
      val j = (math.random * i).toInt
      val cj = cards(j)
      cards(j) = cards(i-1)
      cards(i-1) = cj
    }
  }
  def draw(): Card = {
    assert(count > 0)
    count -= 1
    cards(count)
  }
}
```

Now that we have a working deck, we can write the client code, that is, we can implement the Blackjack game. Here is the beginning of the game (you can download the entire program from the course webpage). Note how easy it is to read this code, because we have hidden all the complexity of storing cards and shuffling and managing the deck inside the `Card` and `Deck` classes.

```
val deck = new Deck()

// initial cards
var player = Array(deck.draw())
println("You are dealt " + player.head)
var dealer = Array(deck.draw())
println("Dealer is dealt a hidden card")

player = player :+ deck.draw()
println("You are dealt " + player.last)
dealer = dealer :+ deck.draw()
println("Dealer is dealt " + dealer.last)
printf("Your total is %d\n",
       handValue(player))
```

11 More constructors?

Sometimes it is useful to have more than one constructor for objects. For instance, our `Point` class above requires us to provide the coordinates of a point. But sometimes we have no useful coordinates, we just want to create some point that can be filled in later. We would just like to say `new Point` to get a new `Point` object, with some default coordinates (both zero, for instance). With the definition above, this doesn't work:

```
scala> class Point(var x: Int, var y: Int)
defined class Point
scala> val p = new Point
error: not enough arguments for
 constructor Point: (x: Int,y: Int)Point.
Unspecified value parameters x, y.
```

The solution is to add a second constructor to the `Point` class. A second constructor is defined like a method with the special name `this`:

```
class Point(var x: Int, var y: Int) {
  def this() = { this(0, 0) }
  override def toString: String =
    "Point(" + x + "," + y + ")"
}
```

Note that the second constructor has no return type (since it is clear that it must create a `Point` object. Inside the second constructor, we must call the *primary constructor* of the class. Again, this looks like a method call with the special name `this`.

We can now create points without arguments:

```
scala> var p = new Point
p: Point = Point(0,0)
scala> p.x = 7
scala> p
res0: Point = Point(7,0)
```

Except to refer to the constructor, the keyword `this` has another use in Scala: Inside a method, `this` is always a name for the current object itself.