Stateful and Stateless Objects

Jacek Wasilewski

Stateful Object

- **Stateful object** – object which state changes over time
- **State** – a set of variables that can be changed in the course of a computation
- An object has state or **is stateful** if its **behavior** is influenced by its history
  - an object is mutable or has a mutable state
- Example: bank account object has state because we might get different answers if we ask "can I withdraw 100 euro?".

Stateful Objects in Scala

- Mutable state is built from variables
- Use **`var`** instead of **`val`**
- To have a state, every variable has to be initialized
- If initializer is missing, it is not treated as a variable definition
  `var x: Int`
  - if that appears in a class, it is treated as an abstract access method
- If you do not know or care about the initializer, use wildcard:
  `var x: T = _`
- Or **`null`** for reference types, **`false`** for Booleans, **`0`** for numbers

Example (1)

- Model a class that represents a bank account
- Bank account stores its balance
- We can deposit some amount of cash
  - only positive amount of cash
- We can withdraw some amount of cash
  - only positive amount of cash
  - not larger than the balance
  - if larger, return an error

Example (2)

```
class BankAccount {
    private var balance = 0
    def deposit(amount: Int) = {
        if (amount > 0)
            balance += amount
    }
    def withdraw(amount: Int): Int = {
        if (0 < amount &&
            amount <= balance) {
            balance -= amount
            balance
        } else {
            sys.error(”no funds")
        }
    }
}
```

- Balance is private
- Balance is a variable
- Deposit modifies balance
- Withdraw modifies balance
- Balance represents our state

Example (3)

```
scala> val account = new BankAccount
account: BankAccount = BankAccount@d44fc21

scala> account deposit 50

scala> account withdraw 20
res1: Int = 30

scala> account withdraw 20
res2: Int = 10

scala> account withdraw 15
java.lang.RuntimeException: no funds
  at scala.sys.package$.error(package.scala:27)
  at BankAccount.withdraw(<console>:22)
  ... 33 elided
```

Stateless Object

- The opposite to stateful object
  - immutable objects/classes
  - simply, objects that cannot be modified
- All of the information is provided when it is created and is fixed for the lifetime of the object
- Example: **`String`** class

Why to use immutable classes?

- Reasons
  - easier to design
  - easier to implement
  - easier to use than mutable classes
  - less prone to errors
  - more secure

Making classes immutable

- Rules (for Java but similar for Scala)
  1. Do not provide any methods that modify the object's state (mutators)
  2. Ensure that class cannot be extended – **`final`** classes
  3. Make all fields final – **`val`** in Scala
  4. Make all fields private
  5. Ensure exclusive access to any mutable components – make sure it is not possible to obtain references to objects that your object uses; make defensive copies in constructors, etc.

Example (1)

```
class PhoneNumber(areaCodeP: Int, prefixP: Int, lineNumberP: Int) {
    var areaCode: Int = areaCodeP
    var prefix: Int = prefixP
    var lineNumber: Int = lineNumberP
}
```

```
scala> val pn = new PhoneNumber(86, 111, 2222)
pn: PhoneNumber = PhoneNumber@1e4f4a5c
```

```
scala> pn.lineNumber
res6: Int = 2222
```

```
scala> pn.lineNumber = 333
pn.lineNumber: Int = 333
```

```
scala> pn.lineNumber
res7: Int = 333
```

Example (2)

```
final class PhoneNumber(areaCodeP: Int, prefixP: Int,
                        lineNumberP: Int) {
    private val areaCode: Int = areaCodeP
    private val prefix: Int = prefixP
    private val lineNumber: Int = lineNumberP

    def getAreaCode: Int = areaCode
    def getPrefix: Int = prefix
    def getLineNumber: Int = lineNumber
}
```

Example (3)

```
scala> val pn = new PhoneNumber(86, 111, 2222)
pn: PhoneNumber = PhoneNumber@3a0baae5

scala> pn.lineNumber
error: value lineNumber in class PhoneNumber cannot be accessed in
PhoneNumber

scala> pn.lineNumber = 333
error: value lineNumber in class PhoneNumber cannot be accessed in
PhoneNumber

scala> pn.getLineNumber
res9: Int = 2222

scala> res9 = 3333
error: reassignment to val
```

Example (4)

```
final class Complex(reP: Double,
                    imP: Double) {
  private val re: Double = reP
  private val im: Double = imP

  def realPart: Double = re
  def imaginaryPart: Double = im

  def add(c: Complex): Complex = {
    new Complex(re + c.re, im + c.im)
  }
  def subtract(c: Complex): Complex = {
    new Complex(re - c.re, im - c.im)
  }
  def multiply(c: Complex): Complex = {
    new Complex(re * c.re - im * c.im,
                re * c.im + im * c.re)
  }
  def divide(c: Complex): Complex = {
    val tmp: Double = c.re * c.re +
                      c.im * c.im
    new Complex((re * c.re + im * c.im) / tmp,
                (im * c.re - re * c.im) / tmp)
    }
}
```

Immutable Objects

- Immutable objects
  - are simple – can be in exactly one state
  - are thread-safe, require no synchronization
  - can be shared freely
- Frequently used immutable objects could be provided as public static final constants – or in accompanying object in Scala
- It is much easier to maintain your code if it uses immutable objects
- Only real disadvantage is that they require a separate object for each distinct value

Immutable vs Mutable

- Classes should be immutable unless there is a very good reason to make them mutable
- For some classes immutability might not be practical
  - if class cannot be immutable, limit its mutability as much as possible.
  - make every field final unless there is a compelling reason to make it non-final