Mutable Objects

Effective Programming in Scala

How To Do Side-Effects?

In the previous lesson, we have seen how to implement a “pure” random number generator.

How can we implement a random number generator that uses side-effects?

var Definitions

We need to use a var definition:

```
object Generator:
  var previous: Int = 42

  def nextInt(): Int =
    val result = previous * 22_695_477 + 1
    previous = result
    result
end Generator
```

var Definitions

We need to use a `var` definition:

```
object Generator:
  var previous: Int = 42

  def nextInt(): Int =
    val result = previous * 22_695_477 + 1
    previous = result
    result
end Generator
```

We say that `Generator` is a **mutable object**. It's internal state may change over time.

Mutable Classes

Here is an example of mutable class that models a bank account:

```
class BankAccount:
  private var balance: Int = 0
  def deposit(amount: Int): Int =
    if amount > 0 then balance = balance + amount
    balance

  def withdraw(amount: Int): Int =
    if amount > 0 && amount <= balance then
      balance = balance - amount
    balance
end BankAccount
```

Bank Account: Usage

```
val account = BankAccount()

account.deposit(30)  // 30
account.deposit(30)  // 60
account.withdraw(50) // 10
```

Identity and Change

Mutable classes raise a new problem: what does it mean for two instances to be equal?

In an immutable world, if we write:

```
val x = Rectangle(5, 8)
val y = Rectangle(5, 8)
```

Then, we expect `x == y` to be true.

Identity and Change (2)

More generally, in an immutable world, if we write:

```
val x = E
val y = E
```

where `E` is an arbitrary expression, then we expect that `x` and `y` are the same. That is to say that we could refactor the program to:

```
val x = E
val y = x
```

without changing the meaning of the program.

Identity and Change (3)

But once we use mutable objects, the situation is different. For example:

```
val x = BankAccount()
val y = BankAccount()
```

Question: Are x and y the same?

0 Yes

0 No

Identity and Change (3)

But once we use mutable objects, the situation is different. For example:

```
val x = BankAccount()
val y = BankAccount()
```

Question: Are x and y the same?

0 Yes

X No

Case Classes vs Plain Classes

By default, Scala already does "the right thing": comparing two instances of `BankAccount` with the same balance will return `false`, but comparing two instances `Rectangle` with the same length and width will return `true`.

This highlights one key difference between plain classes and case classes: plain classes equality is checked by comparing the "identity" of their instances, whereas case classes equality is checked by comparing the values carried by their instances

Reasoning on Mutable Objects

Mutable objects are not "refactoring-proof": we can't replace

```
val x = E
val y = E
```

with

```
val x = E
val y = x
```

without changing the meaning of our program.

Therefore, it is a good practice to prefer immutable data types (for instance, immutable collections).

Summary

The state of mutable objects can change over time.

Consequently, distinct instances of mutable classes are not *the same* even though they carry the same values at one point in time. We say that they have distinct **identities**.

It is a good practice to prefer immutable data types.