EPFL

A Case for Side-Effects

Effective Programming in Scala

Benefits of Side-Effects

Previously, we have seen that side-effecting operations have drawbacks that require developers to manipulate them with extra care.

Why do we have side-effects at all?

Benefits of Side-Effects

Previously, we have seen that side-effecting operations have drawbacks that require developers to manipulate them with extra care.

Why do we have side-effects at all?

Because doing otherwise also has drawbacks...

“Pure” Random Number Generator: Definition

Let us try to implement a random number generator without using side-effects.

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

object Generator:
  def init: Generator = Generator(42)
```

(We use a Linear Congruential Generator.)

“Pure” Random Number Generator: Usage

The operation nextInt returns a random Int value and the next Generator to use:

```
val gen1 = Generator.init
val (x, gen2) = gen1.nextInt()
println(x) // 953210035
val (y, _) = gen1.nextInt()
println(y) // 953210035
val (z, gen3) = gen2.nextInt()
println(z) // -570911984
```

“Pure” Random Number Generator: Usage (2)

The `between` operation can be implemented like this in the class `Generator`:

```scala
def between(x: Int, y: Int): (Int, Generator) =
  val min   = math.min(x, y)
  val delta = math.abs(x - y)
  val (randomValue, nextGenerator) = nextInt()
  ((randomValue % delta) + min, nextGenerator)
```

“Pure” Random Number Generator: Usage (2)

The between operation can be implemented like this in the class Generator:

```
def between(x: Int, y: Int): (Int, Generator) =
  val min   = math.min(x, y)
  val delta = math.abs(x - y)
  val (randomValue, nextGenerator) = nextInt()
  ((randomValue % delta) + min, nextGenerator)
```

Our random number generator can be used like this:

```
val gen1 = Generator.init
val (windowSide, _) = gen1.between(1, 4)  // windowSide = 2
val windowArea = windowSide * windowSide  // : Int = 4
```

“Pure” Random Number Generator

We can now interchangeably use `def` or `val`, this doesn’t affect the behavior of our program!

However, we have to use each `Generator` only once.

The problem becomes worse if several parts of our program use random numbers:

```scala
val gen1 = Generator.init
getSomething(gen1, getSomeOtherThing(gen1))
```

Here, by passing `gen1` at two places (as a parameter of `getSomething` and `getSomeOtherThing`) we will get the same random numbers in both places!

Context Threading

The correct way to use our random number generator is the following:

```
val gen1 = Generator.init
val (otherThing, gen2) = getSomeOtherThing(gen1)
getSomething(gen2, otherThing)
```

The operation `getSomeOtherThing` needs to not only take an instance of `Random` as parameter, but also to return the next instance to use.

When we call `getSomething`, we must be careful to use `gen2` and not `gen1`.

Context Threading

The correct way to use our random number generator is the following:

```
val gen1 = Generator.init
val (otherThing, gen2) = getSomeOtherThing(gen1)
getSomething(gen2, otherThing)
```

The operation `getSomeOtherThing` needs to not only take an instance of `Random` as parameter, but also to return the next instance to use.

When we call `getSomething`, we must be careful to use `gen2` and not `gen1`.

Arguably, this is more tedious to write than using the side-effecting random number generator, `scala.util.Random`:

```
getSomething(getSomeOtherThing())
```

Summary

“Pure” alternatives to side-effects introduce accidental complexity by requiring developers to explicitly carry over the “context” they operate on.

Note that there are techniques to simplify this task, but they come with constraints.

There is no silver bullet. Just use the approach that works best for you by taking into account their pros and cons.