EPFL

Refactoring-Proof Programs

Effective Programming in Scala

Refactoring-Proof Programs

In the previous lecture we saw that it is sometimes difficult to reason about the meaning of programs.

In this lecture we will go one step further and identify two types of programs, and the assumptions we can make (or not) to reason about them.

Puzzle

Consider the following program that computes the area of a square window of a size randomly chosen between the values of 1, 2, or 3 meters.

```scala
def windowSide = util.Random.between(1, 4)
def windowArea = windowSide * windowSide
println(windowArea)
```

(`util.Random.between` is a method provided by the Scala standard library)

What does the expression `windowArea` evaluate to?

Puzzle

Consider the following program that computes the area of a square window of a size randomly chosen between the values of 1, 2, or 3 meters.

```scala
def windowSide = util.Random.between(1, 4)
def windowArea = windowSide * windowSide
println(windowArea)
```

(`util.Random.between` is a method provided by the Scala standard library)

What does the expression `windowArea` evaluate to?

Our expectation is to get either 1, 4, or 9, depending on the random value that `windowSide` got.

However, this is not what we observe.

Stabilizing Expressions Evaluation

The problem is that the expression "`windowSide * windowSide`" evaluates `windowSide` two times, and each evaluation returns a different value.

We can "fix" the problem by using a `val` definition instead of a `def` definition, so that the result of the first evaluation of `windowSide` is reused each time `windowSide` is used.

```
val windowSide = util.Random.between(1, 4)
def windowArea = windowSide * windowSide
```

Refactoring-Proof Programs

Why does changing a `def` into a `val` have such an important impact on the behavior of the program? Is this change not just an optimization (avoiding re-computing the same thing several times)?

Actually, the fact that evaluating the expression “`Random.between(1, 4)`” several times returns possibly different results has important consequences on our ability to reason about it.

For instance, we must be very careful when we refactor this code.

Refactoring-Proof Programs (2)

Consider the following algebraic identity:

```
x + x == 2 * x
```

It suggests that we could refactor any expression of the form "`x + x`" into "`2 * x`" without changing its meaning. For instance:

```
42 + 42 == 2 * 42
```

However:

```
Random.between(1, 4) + Random.between(1, 4) == 2 * Random.between(1, 4)
```

This one might not be true.

Doing vs Describing

What makes `Random` operations different so that we can not apply refactorings to programs that use them?

Doing vs Describing

What makes `Random` operations different so that we can not apply refactorings to programs that use them?

- In addition to returning a value, these operations also “do something” out of the control of our program (in this case, changing the internal state of the `Random` object).

We say that these operations have **side-effects**.

Doing vs Describing (2)

Can you think of other operations that have side-effects?

Doing vs Describing (2)

Can you think of other operations that have side-effects?

We can classify them in two categories:

- operations that modify the state of the program,
- operations that communicate with the "outside world" (printing to a file, reading from a socket, etc.).

Doing vs Describing (2)

Can you think of other operations that have side-effects?

We can classify them in two categories:

- operations that modify the state of the program,
- operations that communicate with the "outside world" (printing to a file, reading from a socket, etc.).

By contrast, operations that only describe what result to return given a set of input parameters are always refactoring-proof. They are also called "purely functional" or "referentially transparent" in the literature.

Doing vs Describing (3)

The example we gave is simplistic and it might seem easy to not fall for it.

However, in large code bases operations with side-effects can be hard to manage because small changes can have large and unexpected consequences.

Referentially transparent operations are not subject to these problems because they do nothing more than returning a result that only depends on their input parameters.

Summary

Operations that have side-effects open the door to a class of issues that do not exist with referentially transparent operations.

Side-effecting operations should be used with extra care.