**FUNCTIONAL PROGRAMMING IN SCALA**

**Introduction to fuctional programming**

**What is functional programming?**

Functional programming (FP) is base don a simple premise with far-reaching implications: we construct our programs using only pure functions, in other words, functions that have **no side effects**. What are side efects? A function has a side efect if it does somthing or other than simply return a result, for example:

.Modifying a variable

.Modifying a data structure in place

.Setting a fiel don an object

.Throwing an exception or halting with an error

.Printing to the console or Reading user input

.Reading from or writing to a file

.Drawing on the screen

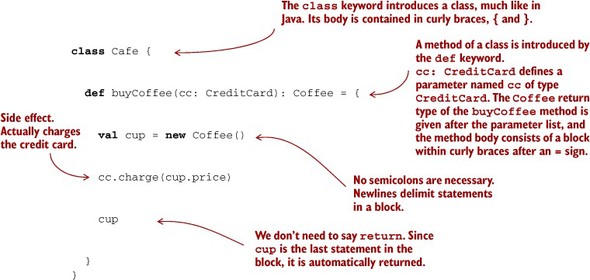
It may be difficult to imagine. How is it even posible to write useful programs at all? If we can’t reassign variables, how do we write simple programs like loops?

What about working with data that changes, or handling errors without throwing exceptions? How can we write programs that must perform I/O. The answer is that FP is a restriction on how we write programs, but not on what programs we can express, we’ll learn how to express all of our programs without these side effects.

Lets look at an example that demonstrates some of the benefits of programming with pure functions.

**A program with side effects**

Suppose we’re implementing a program to handle purchases at a coffe shop. We’ll begin with a Scala program that uses side effects in its implementation (also called **impure program**).



The line cc.charge(cup.price)is an example of a side effect. Charging a credit card involves some interaction with the outside world, suppose it rrequires contacting the credit card Company via som web service, autorizing the transaction, charging the card, and (if succesful) parsisting some record of the transaction for later reference. Bu tour function merely returns a Coffe and these other actions are happening on the side, hence the term “side effect”. As a result of this side effect, the code is difficult to test. We don’t want our test to actually contat the credit card and Company and charge the card! THis lack of testability is suggesting a design change: arguably, CreditCard shouldn’t have any knowledge baked into it about how to contact the credit card Company to actually execute a charge, nor should it have knowlege of how to persist a record of this charge in our internal systems. We can make the code more modular and testable by letting CreditCard be ignorant of these concerns and passing a Payments object into buyCoffe.

**class** Cafe {

**def** buyCoffee(cc: CreditCard, p: Payments): Coffee = {

**val** cup = **new** Coffee()

p.charge(cc, cup.price)

cup

}

}

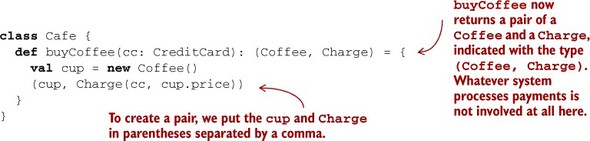
Though side effect still occur when we call p.charge(cc, cup.price), we have at least regained some testability. Payments can be an interface, and we can wrrite a mock implementation of this interfece that is suitable for testing. But that isn’t ideal either. We’re forced to make Payments an interface, when a concrete class may have been fine otherwise, and any mock implementaton will be awkward to use.

Separate from the concer of testing, there´s another problema: it’s difficult to reuse buyCoffe. Suppose a customer, Alice, would like to order 12 cups of coffe. Ideally we could just reuse buyCoffee for this, perhaps calling it 12 times in a loop. But as it is currently implemend that involve contacting the payment system 12 times, authorizing 12 separate charges to Alice’s credit card! That adds more procesing fees and isn’t good for Alice or the coffe shop.

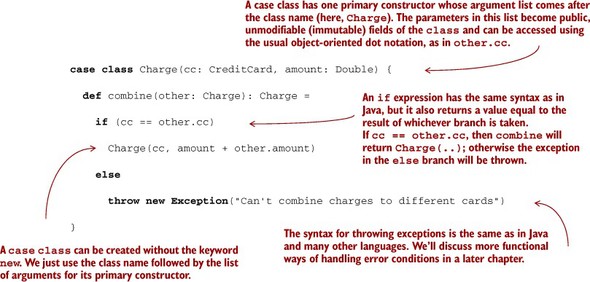
What can we do about this? We could write a whore new function, buyCoffees, with special logic for batching up the charges. Her that might not be such a deal, sincce the logic of buy coffe is so simple, but in other cases the logic we need to duplicate may be nontrivial.

**A functional solution: removing the side effects**

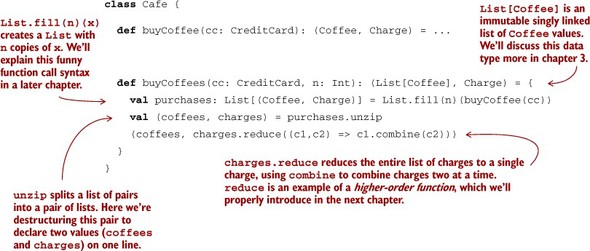
The functional solition is to eliminate side effects and have buyCoffee return the charge as a value in addition to returning the Coffee. The concerns of processing the charge by sending it off to the credit card Company, persisting a record of it, and so on, will be handled elsewhere. Again, we’ll cover Scala’s syntex more in later chapters, but here’s what a functional solution might look like:



Here we’ve separated the concern of creating a charge from the processing or interpretation of that charge. The buyCoffee function now returns a Charge as a value along with the Coffee. We’ll see shortly how this lets us reuse it more easily to purchase multiple coffees with a single transaction. But what is Charge? It’s a data type we just invented containing a CreditCard and an amount, equipped with a handy function, combine, for combining charges with the same CreditCard:



Now let’s take a look at buyCofees, to implement the purchase of n cups of coffee. Unlike before, this can now be implemented in terms of buyCoffee, as we had hoped.



Overall, this solution is a marked improvement—we’re now able to reuse buyCoffee directly to define the buyCoffees function, and both functions are trivially testable without having to define complicated mock implementations of some Payments interface! In fact, the Cafe is now completely ignorant of how the Charge values will be processed. We can still have a Payments class for actually processing charges, of course, but Cafe doesn’t need to know about it.

**def** coalesce(charges: List[Charge]): List[Charge] = charges.groupBy(\_.cc).values.map(\_.reduce(\_ combine \_)).toList

We’re passing functions as values to the groupBy, map, and reduce methods. You’ll learn to read and write one-liners like this over the next several chapters. The \_.cc and \_ combine \_ are syntax for **anonymous functions**, which we’ll introduce in the next chapter. This function takes a list of charges, groups them by the credit card used, and then combines them into a single charge per card. It’s perfectly reusable and testable without any additional mock objects or interfaces. Imagine trying to implement the same logic with our first implementation of buyCoffee!

**Exactly what is a pure function?**

A function f with input type A and output type B (written in Scala as a single type: A => B, pronounced “A to B” or “A arrow B”) is a computation that relates every value a of type A to exactly one value b of type B such that b is determined solely by the value a. Any changing state of an internal or external process is irrelevant to computing the result f(a). For example, a funtion intToString having type Int => String will take every integer to a corresponding string. Furthermore, if it is really a function, it will do nothing else. In other words, a function has no observable effect on the execution of the program other than compute a result given its inputs, we say that it has no side effects.

We can formalize this idea of pure functions using the concept of **referential transparency (RT)**. This is a property of **expressions** in general and not just functions. For the purposes of our discussion, consider an expression to be any part of a program that can be evaluated to a result, anything that you could type into the Scala interpreter and get an answer. For example 2+ 3 is an expression that applies the pure function + to the values 2 and 3 (which are also expressions). This has no side effect. The evaluation of this expression results in the same value 5 every time.

This isa ll it means for an expression to be referentially transparent, in any program, the expression can be replaced by its result without changing any meaning of the program. And we say that a function is **pure** if calling it with RT arguments it is also RT.

**Definition:** An expression e is referentially transparent if, for all programs p, all ocurrences of e in p can be replaced by the result of evaluating e without affecting the meaning of p. A function f is *pure* if the expression f(x) is *referentially transparent* for all *referentially transparent* x.

**Referential transparency, purty, and the substitution model**

Let’s see how the definition of RT applies to our original buyCoffee example:

**def** buyCoffee(cc: CreditCard): Coffee = {

**val** cup = **new** Coffee()

cc.charge(cup.price)

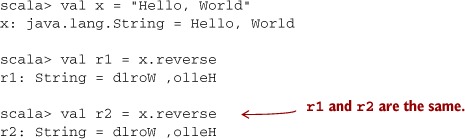
cup

}

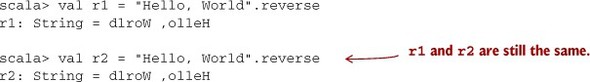
Whatever the return type of cc.charge(cup.price), it’s discarded by buyCoffee. Thus, the result of evaluating buyCoffee(aliceCreditCard) will be merely cup, which is equivalent to a new Coffee(). For buyCoffee to be pure, by our definition of RT, it must be the case that p(buyCoffee(aliceCreditCard)) behaves the same as p(new Coffee()), for any p.

Referential transparency forces the invariant that everything a function does is represented by the value that it returns, according to the result type of the function. This constraint enables a simple and natural mode of reasoning about program evaluation called the **substitution model**. When expressions are *referentially transparent*, we can imagine that computation proceeds much like we’d solve an algebraic equation. We fully expand every part of an expression, replacing all variables with their referents, and then reduce it to its simplest form. At each step we replace a term with an equivalent one; computation proceeds by substituting equals for equals. In other words, RT enables equational reasoning about programs.

Let’s look at two examples, one where all expressions are RT and can be reasoned about using the substitution model, and one where some expressions violate RT, by using the Scala interpreter REPL (Read-Evaluate-Print-Loop).



Supose we replace all occurrences of the term x with the expression referenced by x(its definition as follows).



This transformation doesn’t affect the outcome. The values of r1 and r2 are the same as before, so x was referentially transparent. What’s more, r1 and r2 are referentially transparent as well, so if they appeared i some other part of a larger program, they could in turn be replaced with their values throughout and it would have no effect on the program.

Now, let’s lok at a function that is not referentially transparent. Consider the append function on the java.lang.StringBuilder class. This function operates on the stringBuilder in place. The previous state of the StringBuilder is destroyed after a call to append. Let’s try this out:



So far so good. Now let’s see how this side effect breaks RT. Suppose we substitute the call to append like we did earlier, replacing all occurrences of y with the expression referenced by y:



This transformation of the program results in a different outcome. We therefore conclude that StringBuilder.append is not a pure function. What’s going on here is that although r1 and r2 look like they’re the same expression, they are in fact referencing two different values of the same StringBuilder. By the time r2 calls x.append, r1 will have already mutated the object referenced by x. If this seems difficult to think about, that’s because it is. Side effects make reasoning about program behavior more difficult.

Conversely, the substitution model is simple to reason about since effects of evaluation are purely local (they affect only the expression being evaluated) and we need not mentally simulate sequences of state updates to understand a block of code. Understanding requires only local reasoning. We need not mentally track all the state changes that may occur before or after our function’s execution to understand what our function will do; we simply look at the function’s definition and substitute the arguments into its body. Even if you haven’t used the name “substitution model,” you have certainly used this mode of reasoning when thinking about your code.

This way gives insight into why functional programs are often more modular. Modular programs consist of components that can be understood and reused independently of the whole, such that the meaning of the whole depends only on the meaning of the components and the rules governing their composition; that is, they are composable.