Functions and State

- In a reactive system of any size, there will be sooner or later some state that needs to be changed and maintained, or some state changes that need to be signaled and propagated
- To express this, we are going to broaden now our notion of functions to work together with mutable state
- We will see that these combinations have quite a few repercussions. On the one hand, it gives us powerful new ways to express ourselves in certain categories of systems. On the other hand, it poses considerable challenges how to reason about the resulting systems
- Until now, our programs have been side-effect free. Therefore, the concept of time wasn't important
- For all programs that terminate, any sequence of actions would have given the same result. This is also reflected in the substitution model of computation

Substitution Model

- Programs can be evaluated by rewriting
- The most important rewrite rule covers function applications:

  def f(x_1, … x_n) = B; … f(v_1, … v_n) => def f(x_1, … x_n) = B; … [v_1/x_1, … v_n/x_n]B
- **Rewriting example**: Say you have the following two functions:

  def iterate(n: Int, f: Int => Int, x: Int) =
  *** if (n == 0) x else iterate(n – 1, f, f(x))***
  def square(x: Int) = x * x

  Then the call ***iterate(1, square, 3)*** gets rewritten as follows:
  - ⇨ *if(1 == 0) 3 else iterate(1-1, square, square(3))*
  - ⇨ *iterate(0, square, square(3))*
  - ⇨ *iterate(0, square, 3 * 3)*
  - ⇨ *iterate(0, square, 9)*
  - ⇨ *if (0 == 0) 9 else iterate(0-1, square, square(9))*
  - ⇨ *9*
- **Observation**: Rewriting can be done anywhere in a term, and all rewritings which terminate lead to the same solution. This is an important result of the λ-calculus, the theory behind functional programming. For instance, for the expression:

  if (1 == 0) 3 else iterate(1 – 1, square square(3))

  we can rewrite the expression on two ways:

  iterate(0, square, square(3))

  or

  if (1 == 0) 3 else iterate(1 – 1, square, 3 * 3)

  And the important part is that it doesn't matter which of the two we pick, because in the end, both will give the same answer, 9.
- The idea that we can rewrite anywhere in a term but all results yield the same result, is sometimes called confluence. This confluence is also called the Church-Rosser Theorem of lambda calculus.
- All these observations hold only in the world of pure functional programming

Stateful Objects

- One normally describes the world as a set of objects, some of which have state that changes over the course of time
- An object has a state if its behavior is influenced by its history
- **Example**: a bank account has a state, because the answer to the question "can I withdraw 100 CHF ?" may vary over the course of the lifetime of the account

Implementation of State

- Every form of mutable state is constructed from variables
- A variable definition is written like a value definition, but with the keyword **var** in place of val:

```
var x: String = "abc"
var count = 111
```

- Just like a value definition, a variable definition associates a value with a name
- However, in the case of variable definitions, this association can be changed later through an **assignment**, like in Java:

```
x = "hi"
count = count + 1
```

State in Objects

- In practice, objects with state are usually represented by objects that have some variable members
- **Example**: here is a class modeling a bank account

```
class BankAccount {
    private var balance = 0
    def deposit(amount: Int): Unit = {
        if (amount > 0) balance = balance + amount
    }
    def withdraw(amount: Int): Int = {
        if (0 < amount && amount <= balance) {
           balance = balance - amount
        }
        else
            throw new Error("insufficient funds")
    }
}
```

- The class ***BankAccount*** defines a variable balance that contains the current balance of the account
- The methods ***deposit*** and ***withdraw*** change the value of the balance through assignments
- Note that balance is private in the ***BankAccount*** class, therefore it cannot be accesed from outside the class
- To create bank accounts, we use the usual notation for object creation:

```
val account = new BankAccount
```

Working with Mutable Objects

- Here are some lines that manipulate a bank account:

```
val account = new BankAccount
account deposit 50
account withdraw 20 // account has 30 left
account withdraw 20 // account has 10 left
account withdraw 15 // error: insufficient funds
```

- Applying the same operation to an account twice in a row produces different results, so clearly, accounts are stateful objects

Statefulness and Variables

- Consider the definition of the following function:

```
def cons[T](hd: T, tl: => Stream[T]) = new Stream[T] {
    def head = hd
    private var tlOpt: Option[Stream[T]] = None
    def tail: T = tlOpt match {
        case Some(x) => x
        case None => tlOpt = Some(tl); tail
    }
}
```

Question: Is the result of cons a stateful object?

Both yes and no are valid answers, depending on what assumptions you make on the rest of your system. One common assumption is that streams should only be defined over purely functional computations. So the tail operation here should not have a side effect. In that case, the optimization to cache the first value of tlOpt and reuse in on all previous calls to tail is purely a optimization that avoids computations, but that does not have an observable effect outside the class of streams. So the answer would be that, streams are not stateful objects.

On the other hand, if you allow side effect in computations for tail, let's say tail could have a printing statement, then you would see that the second time tail is caught in this string. It would come straight out of the cache, so there would be no side effect performed, including the printing statement. So that means clearly the operation tail depends on the previous history of the object. It would be different depending on whether a previous tail was performed or not. So in that sense, the answer would be that cons is a stateful object, provided that you also allow imperative side effect in computations for tail.

- Consider the following class:

```
class BankAccountProxy(ba: BankAccount) {
    def deposit(amount: Int): Unit = ba.deposit(amount)
    def withdraw(amount: Int): Int = ba.withdraw(amount)
}
```

Question: Are instances of BankAccountProxy stateful objects?

So here the answer is actually more clear cut, even though BankAccountProxy doesn't contain any variable, its behaviour is clearly stateful because it depends on the history. All that it does is that it forwards to this other bank account. So for instance, calling withdraw twice would give you different results, just as the original withdraw would have done. So clearly the bank account proxies are stateful objects.