FUNCTIONAL PROGRAMMING WITH SCALA

By Engr. Farhan Ahmed Karim

Learning-Journey-02

Programming Paradigms

- Programming Paradigm is a approach to programming
- It will be based on set of principles or theory
- Different ways of thinking.
- Different types
  - Imperative
  - Functional

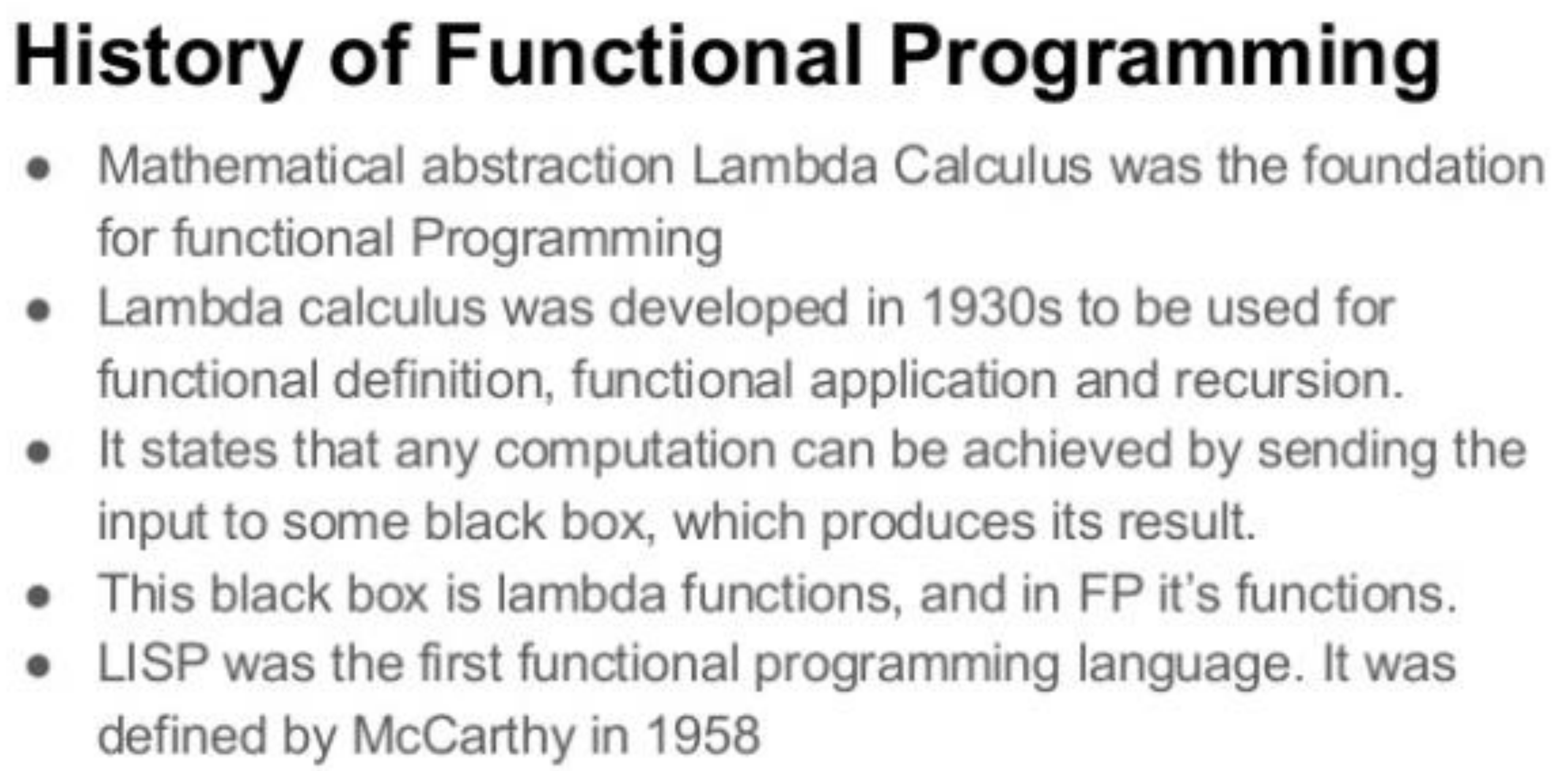
Imperative Programming

- Imperative programming is a programming paradigm that uses statements that change a program's state.
- Example :
  - C#
  - C++
  - Java

Functional Programming

- A programming style that models computation as the evaluation of expression.
- Example:
- ● Haskell
- ● Erlang
- ● Lisp
- ● Clojure

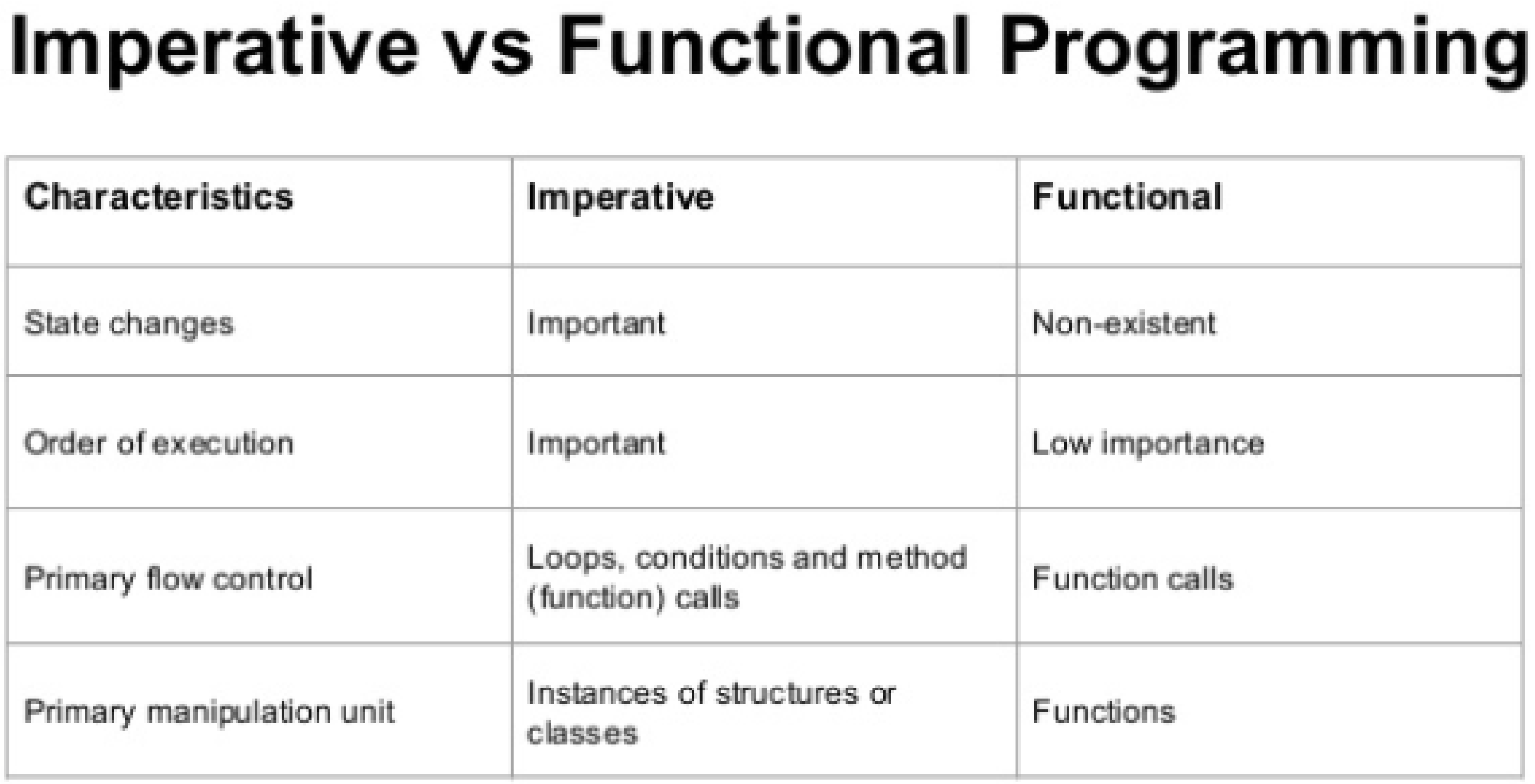
Functional Programming



History of Functional Programming

- Mathematical abstraction Lambda Calculus was the foundation for functional Programming
- Lambda calculus was developed in 1930s to be used for functional definition, functional application and recursion.
- It states that any computation can be achieved by sending the input to some black box, which produces its result.
- This black box is lambda functions, and in FP it's functions.
- LISP was the first functional programming language. It was defined by McCarthy in 1958

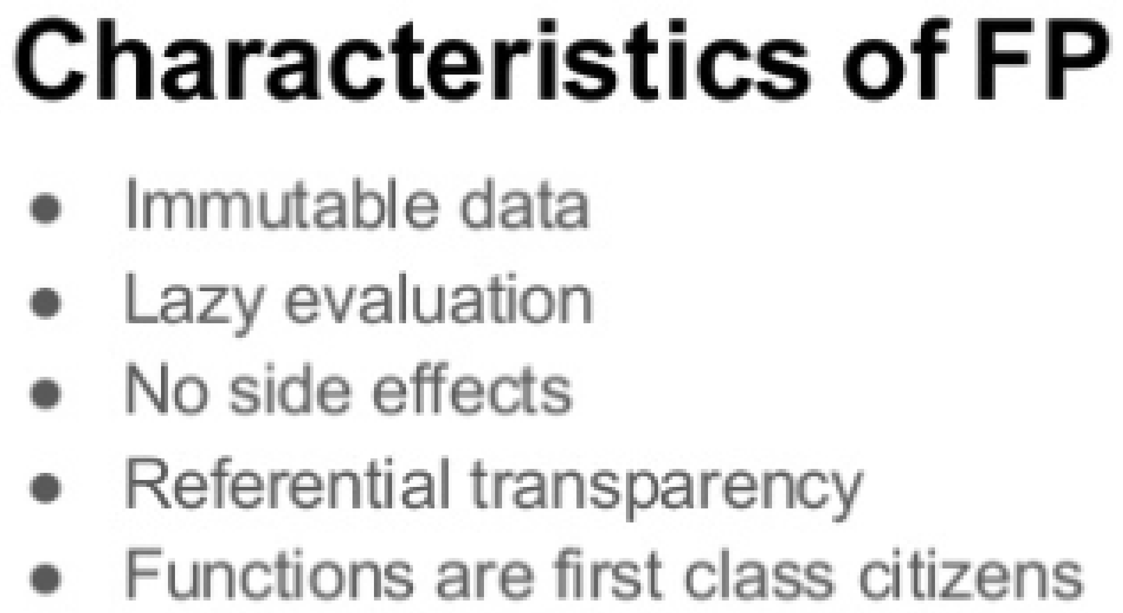
Functional Programming



Imperative vs Functional Programming

Characteristics	Imperative	Functional
State changes	Important	Non-existent
Order of execution	Important	Low importance
Primary flow control	Loops, conditions and method (function) calls	Function calls
Primary manipulation unit	Instances of structures or classes	Functions

Functional Programming



Characteristics of FP

- Immutable data
- Lazy evaluation
- No side effects
- Referential transparency
- Functions are first class citizens

Functional programming: An Introduction

- Functional programming (FP) is based on 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 effects? A function has a side effect if it does something other than simply return a result, for example:

- Modifying a variable
- Modifying a data structure in place
- Setting a field on 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

Question???

- What programming would be like without the ability to do these things, or with significant restrictions on when and how these actions can occur. It may be difficult to imagine.
- How is it even possible 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, like drawing to the screen or reading from a file?
- **The answer is** that functional programming is a restriction on how we write programs, but not on what programs we can express.

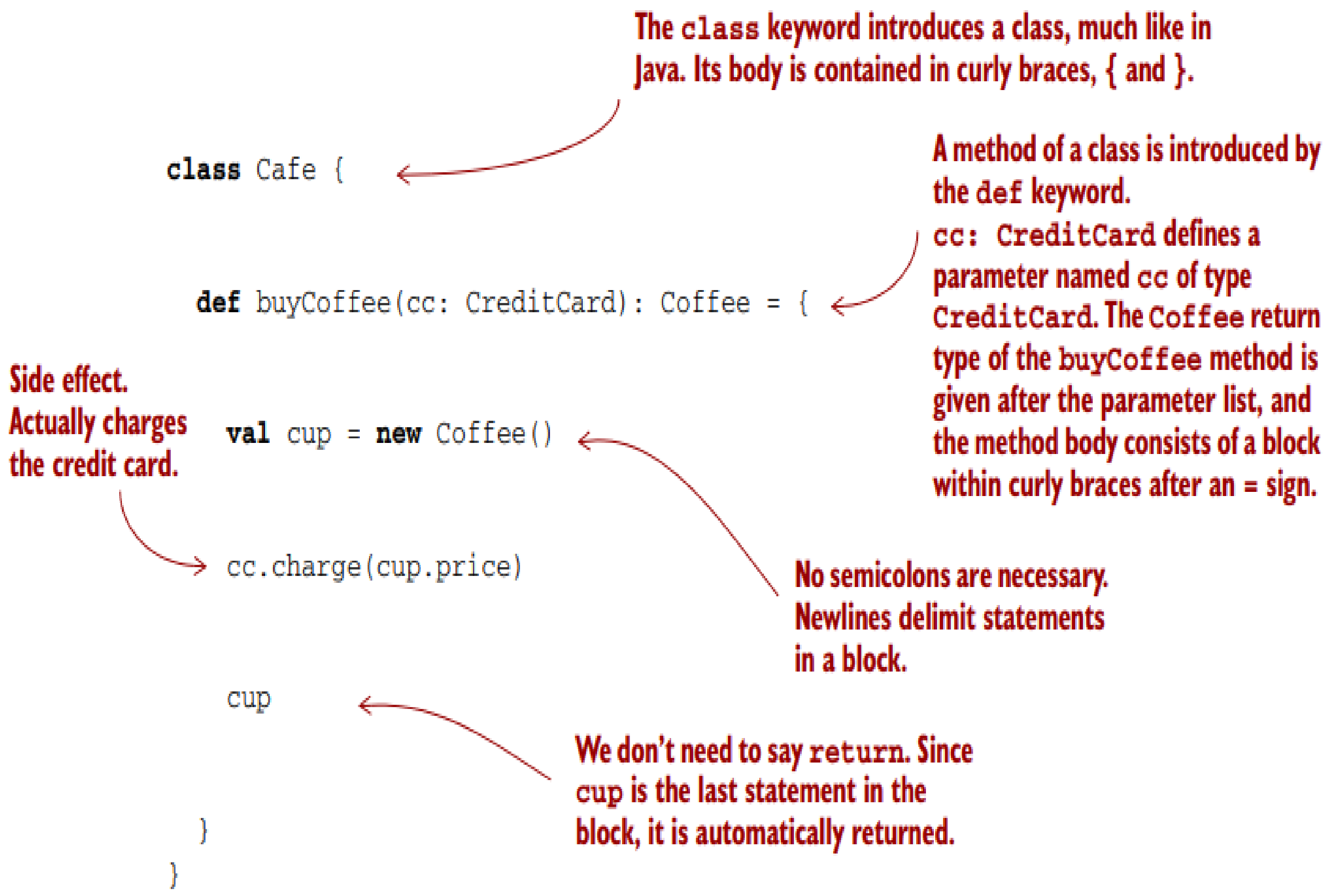
The benefits of FP: a simple example

- Let’s look at an example that demonstrates some of the benefits of programming with pure functions.

- Note: Don’t worry too much about following every detail about code . As long as you have a basic idea of what the code is doing, that’s what’s important.

A program with side effects

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



```
class Cafe {

  def buyCoffee(cc: CreditCard): Coffee = {

    val cup = new Coffee()

    cc.charge(cup.price)

    cup

  }
}
```

The class keyword introduces a class, much like in Java. Its body is contained in curly braces, { and }.

A method of a class is introduced by the def keyword. cc: CreditCard defines a parameter named cc of type CreditCard. The Coffee return type of the buyCoffee method is given after the parameter list, and the method body consists of a block within curly braces after an = sign.

Side effect. Actually charges the credit card.

No semicolons are necessary. Newlines delimit statements in a block.

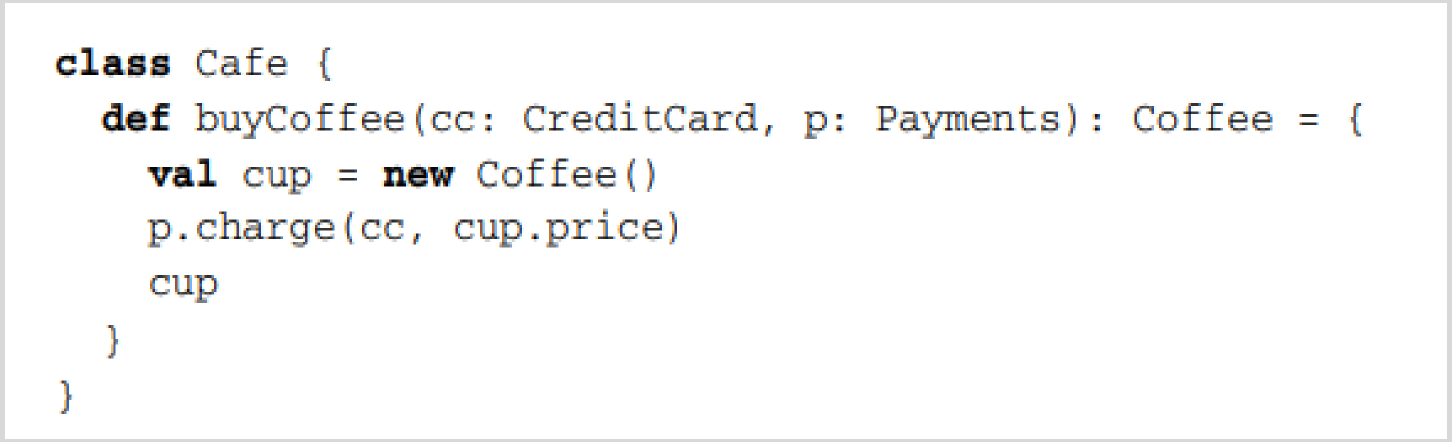
We don't need to say return. Since cup is the last statement in the block, it is automatically returned.

The benefits of FP: a simple example

- 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 requires contacting the credit card company via some web service, authorizing the transaction, charging the card, and (if successful) persisting some record of the transaction for later reference. But our function merely returns a Coffee 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 tests to actually contact the credit card company and charge the card! This lack of testability is suggesting a design change:

Cntd.

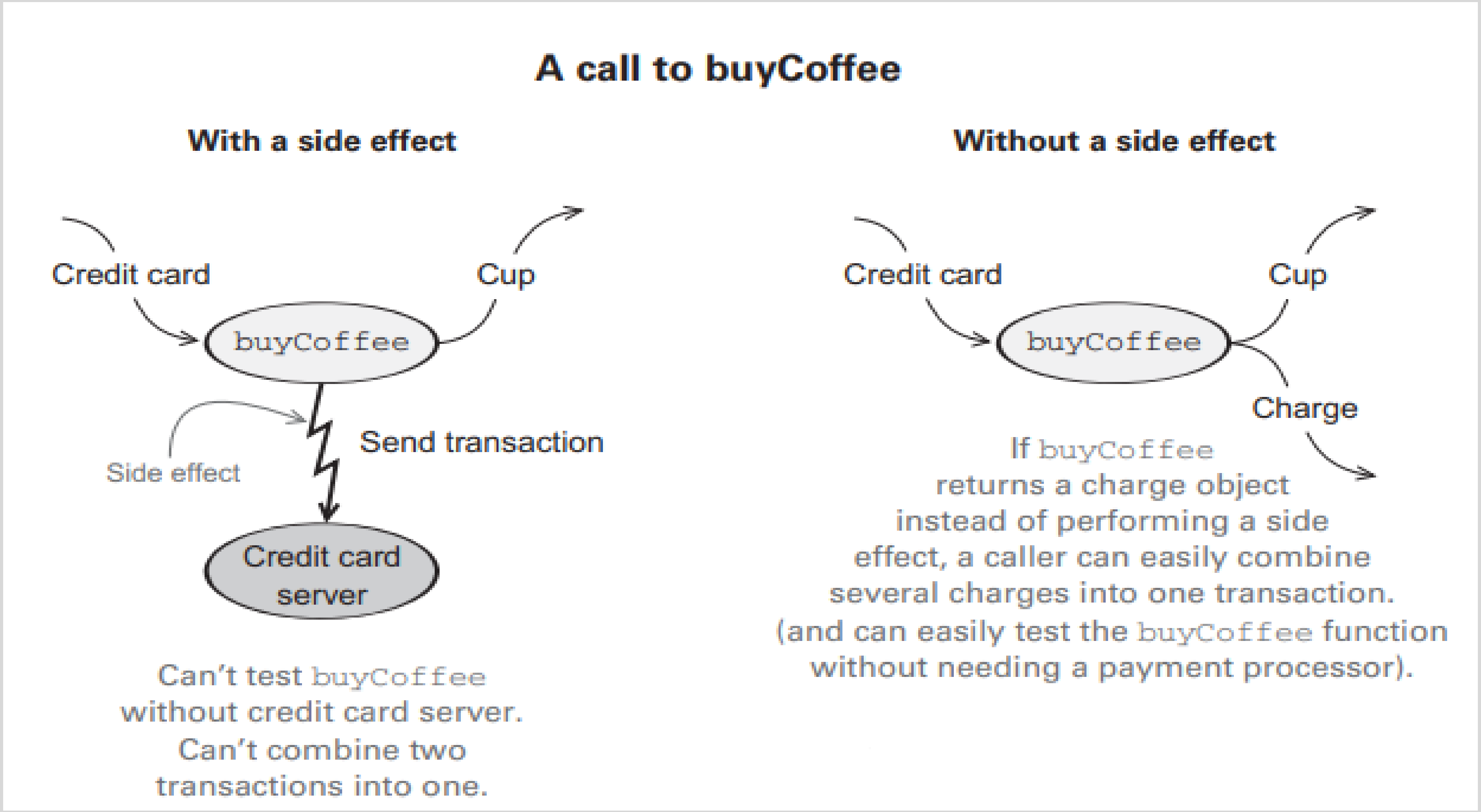
- As a result of this side effect, the code is difficult to test. We don’t want our tests to actually contact the credit card company and charge the card! This lack of testability is suggesting a design change: arguably, CreditCard shouldn’t have any knowledge baked into it about how to contact the credit card company to actually execute a charge, nor should it have knowledge 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 buyCoffee.

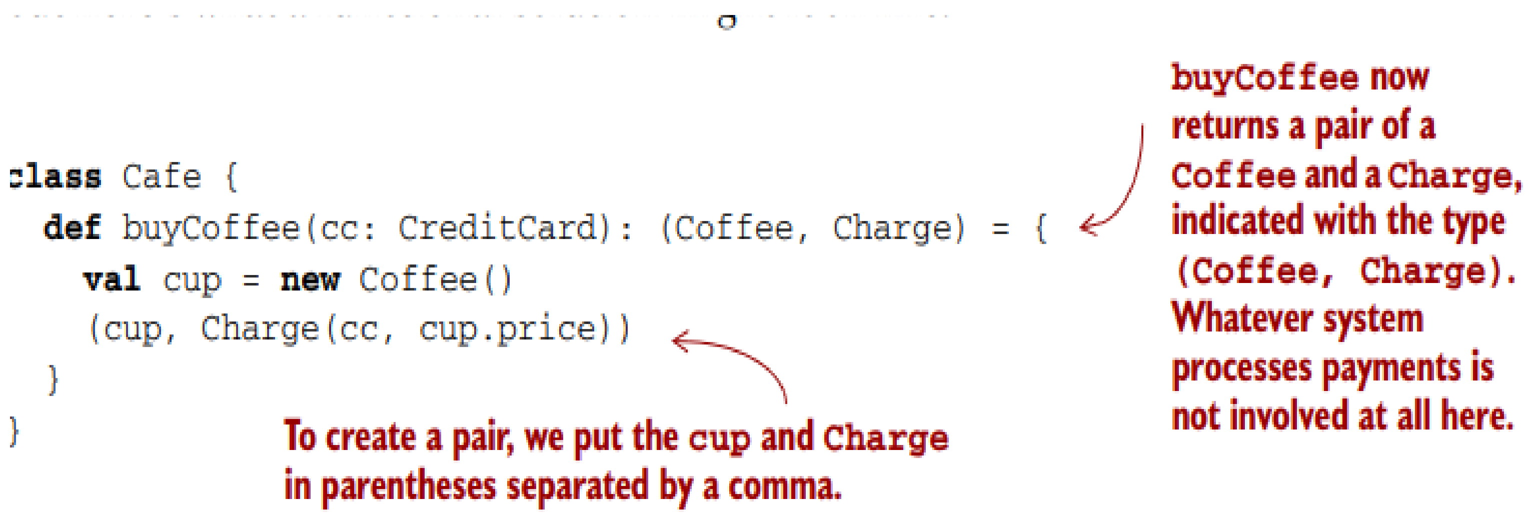


```
class Cafe {
  def buyCoffee(cc: CreditCard, p: Payments): Coffee = {
    val cup = new Coffee()
    p.charge(cc, cup.price)
    cup
  }
}
```

Cntd.

- Though side effects 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 write a mock implementation of this interface 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 implementation will be awkward to use.
- For example, it might contain some internal state that we'll have to inspect after the call to buyCoffee, and our test will have to make sure this state has been appropriately modified (mutated) by the call to charge.
- We can use a mock framework or similar to handle this detail for us, but this all feels like overkill if we just want to test that buyCoffee creates a charge equal to the price of a cup of coffee.


A call to buyCoffee
With a side effect
Credit card
Cup
buyCoffee
Side effect
Send transaction
Credit card server
Can't test buyCoffee without credit card server. Can't combine two transactions into one.
Without a side effect
Credit card
Cup
buyCoffee
Charge
If buyCoffee returns a charge object instead of performing a side effect, a caller can easily combine several charges into one transaction. (and can easily test the buyCoffee function without needing a payment processor).



```
class Cafe {
  def buyCoffee(cc: CreditCard): (Coffee, Charge) = {
    val cup = new Coffee()
    (cup, Charge(cc, cup.price))
  }
}
```

buyCoffee now returns a pair of a Coffee and a Charge, indicated with the type (Coffee, Charge). Whatever system processes payments is not involved at all here.

To create a pair, we put the cup and Charge in parentheses separated by a comma.

A functional solution: removing the side effects

The benefits of FP: a simple example

A case class has one primary constructor whose argument list comes after the class name (here, `Charge`). The parameters in this list become public, unmodifiable (immutable) fields of the `class` and can be accessed using the usual object-oriented dot notation, as in `other.cc`.

```scala
case class Charge(cc: CreditCard, amount: Double) {
  def combine(other: Charge): Charge =
    if (cc == other.cc)
      Charge(cc, amount + other.amount)
    else
      throw new Exception("Can't combine charges to different cards")
}
```

An `if` expression has the same syntax as in Java, but it also returns a value equal to the result of whichever branch is taken. If `cc == other.cc`, then `combine` will return `Charge(..)`; otherwise the exception in the `else` branch will be thrown.

A `case class` can be created without the keyword `new`. We just use the class name followed by the list of arguments for its primary constructor.

The syntax for throwing exceptions is the same as in Java and many other languages. We'll discuss more functional ways of handling error conditions in a later chapter.

Exactly what is a (pure) function?

- We said earlier that FP means programming with pure functions, and a pure function is one that lacks side effects. In our discussion of the coffee shop example, we worked off an informal notion of side effects and purity. Here we’ll formalize this notion, to pinpoint more precisely what it means to program functionally. This will also give us additional insight into one of the benefits of functional programming: pure functions are easier to reason about.

Cntd.

- You should be familiar with a lot of pure functions already. Consider the addition (+) function on integers. It takes two integer values and returns an integer value. For any two given integer values, it will always return the same integer value. Another example is the length function of a String in Java, Scala, and many other languages where strings can’t be modified (are immutable). For any given string, the same length is always returned and nothing else occurs.

Referential Transparency

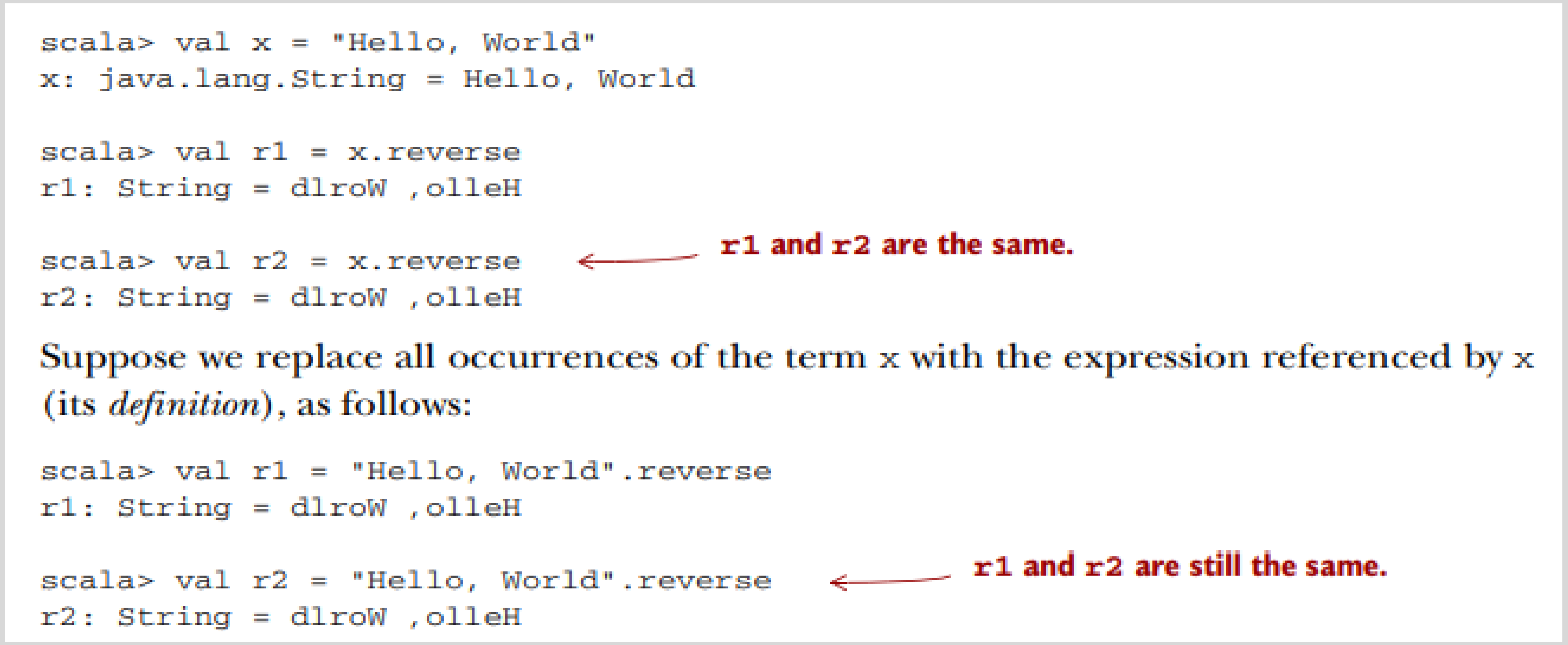
- We can formalize this idea of pure functions using the concept of referential transparency (RT).
- This is a property of expressions in general and not just functions. For the purposes of our discussion, consider an expression to be any part of a program that can be evaluated to a result—anything that 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. In fact, if we saw 2 + 3 in a program we could simply replace it with the value 5 and it wouldn’t change a thing about the meaning of our program

Cntd.

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

Referential transparency, purity, and the substitution model

- Let’s look at two more examples—one where all expressions are RT and can be reasoned about using the substitution model, and one where some expressions violate RT. There’s nothing complicated here; we’re just formalizing something you likely already understand



```
scala> val x = "Hello, World"
x: java.lang.String = Hello, World

scala> val r1 = x.reverse
r1: String = dlroW ,olleH

scala> val r2 = x.reverse        <- r1 and r2 are the same.
r2: String = dlroW ,olleH
```

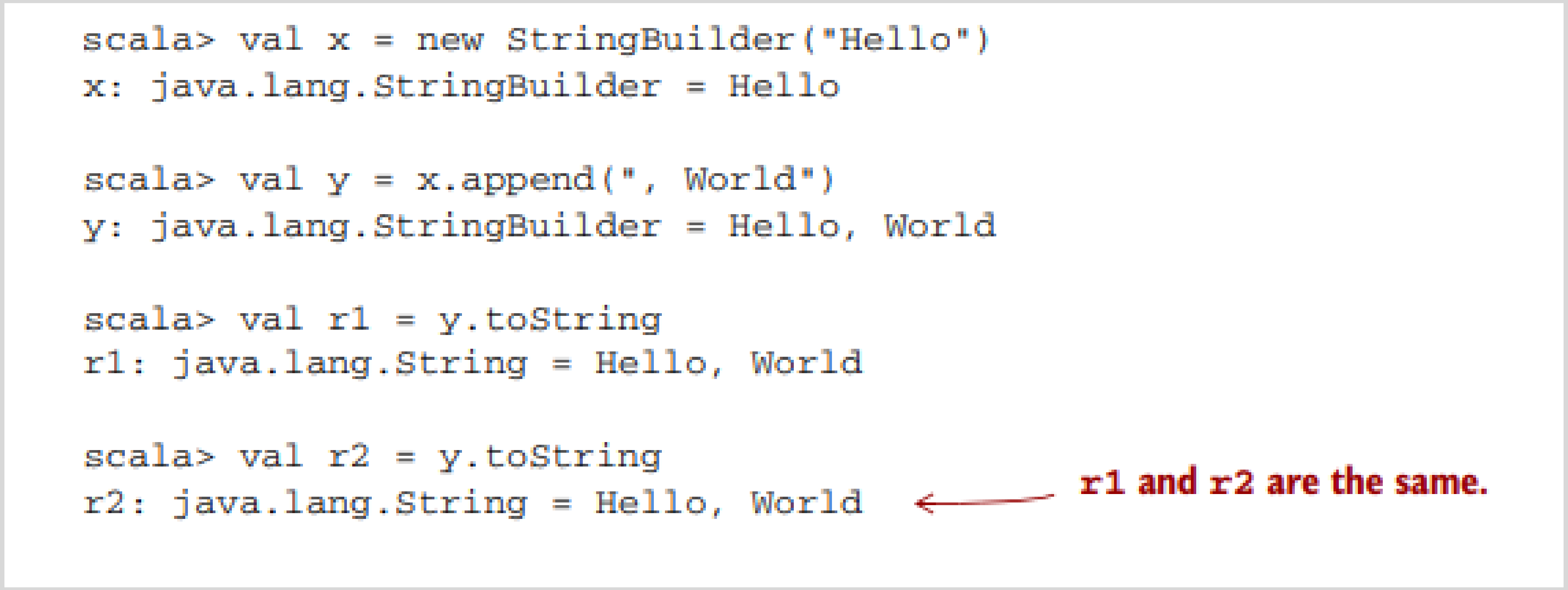
Suppose we replace all occurrences of the term x with the expression referenced by x (its *definition*), as follows:

```
scala> val r1 = "Hello, World".reverse
r1: String = dlroW ,olleH

scala> val r2 = "Hello, World".reverse        <- r1 and r2 are still the same.
r2: String = dlroW ,olleH
```

Cntd

- 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 in 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 look 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:

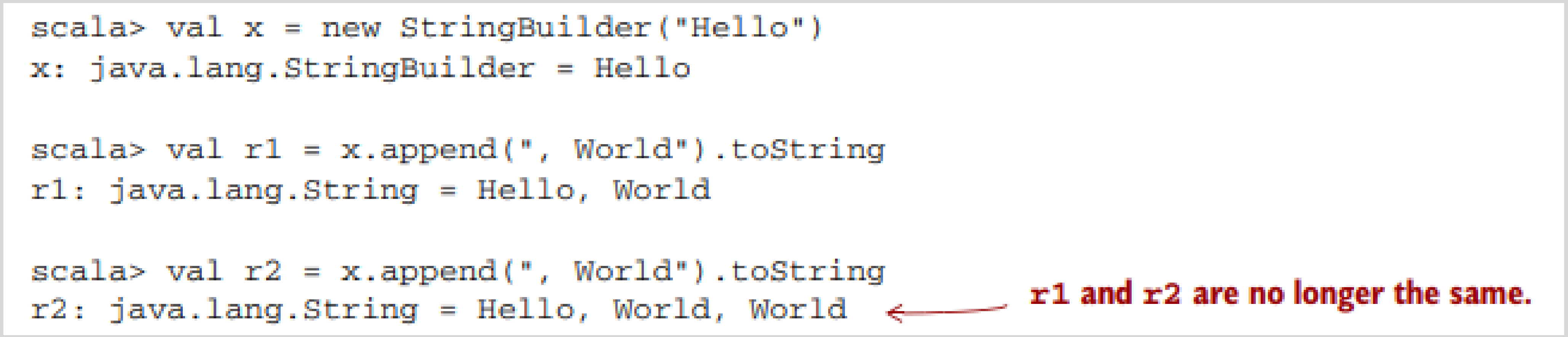


```
scala> val x = new StringBuilder("Hello")
x: java.lang.StringBuilder = Hello

scala> val y = x.append(", World")
y: java.lang.StringBuilder = Hello, World

scala> val r1 = y.toString
r1: java.lang.String = Hello, World

scala> val r2 = y.toString
r2: java.lang.String = Hello, World
```

r1 and r2 are the same.

Cntd.

- 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:



```
scala> val x = new StringBuilder("Hello")
x: java.lang.StringBuilder = Hello

scala> val r1 = x.append(", World").toString
r1: java.lang.String = Hello, World

scala> val r2 = x.append(", World").toString
r2: java.lang.String = Hello, World, World
```

r1 and r2 are no longer the same.

This transformation of the program results in a different outcome. We therefore conclude that StringBuilder.append is not a pure function.

End of Lecture-01

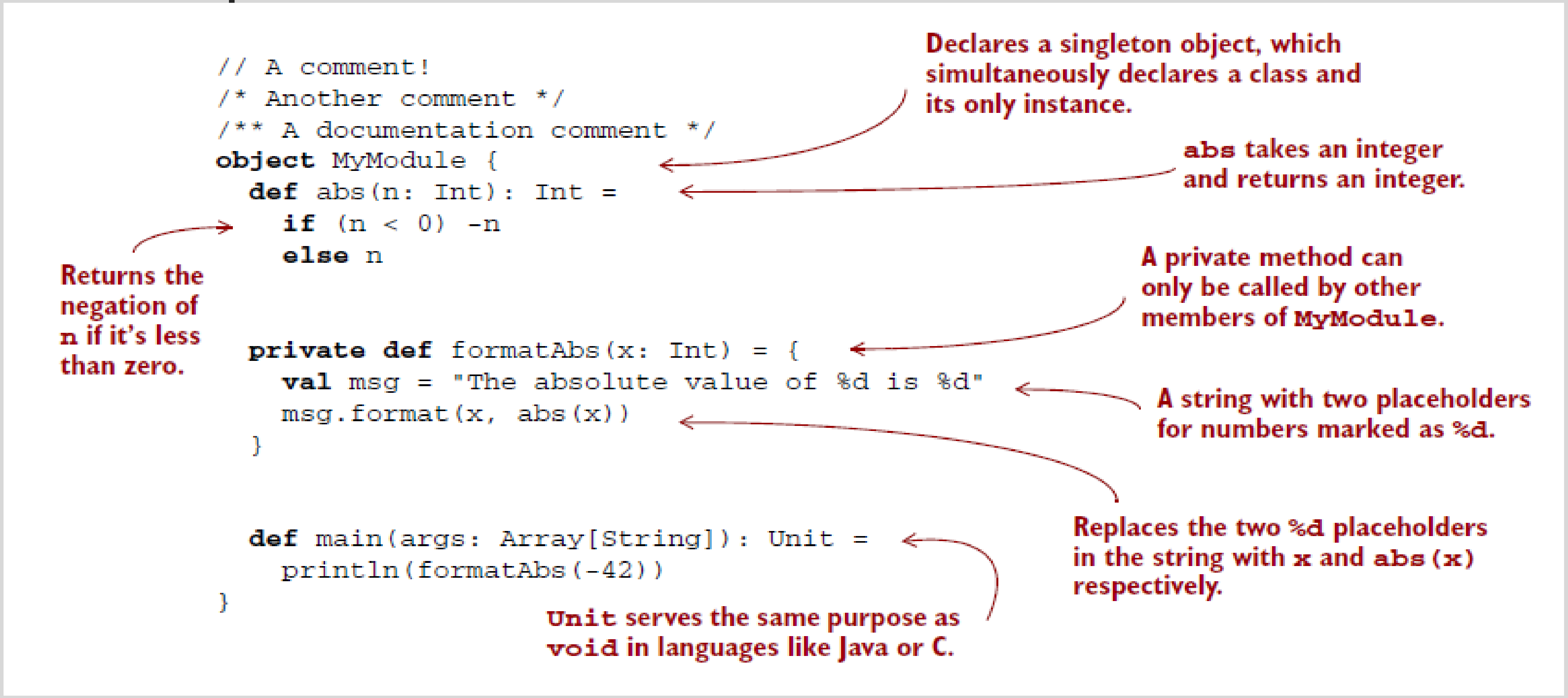
Lecture-02---Getting started with functional programming in Scala

- This Lecture is mainly intended for those students who are new to Scala, to functional programming, or both. Immersion is an effective method for learning a foreign language, so we’ll just dive in. The only way for Scala code to look familiar and not foreign is if we look at a lot of Scala code. We’ve already seen some in the first chapter.
- In this lecture, we’ll start by looking at a small but complete program. We’ll then break it down piece by piece to examine what it does in some detail, in order to understand the basics of the Scala language and its syntax. Our goal is to teach functional programming, but we’ll use Scala as our vehicle, and need to know enough of the Scala language and its syntax to get going.

Cntd..

- We’ll discuss how to write loops using *tail recursive functions*, and we’ll introduce *higher order functions (HOFs)*. HOFs are functions that take other functions as arguments and may themselves return functions as their output. We’ll also look at some examples of *polymorphic* HOFs where we use types to guide us toward an implementation.

Introducing Scala the language: an example



```scala
// A comment!
/* Another comment */
/** A documentation comment */
object MyModule {
  def abs(n: Int): Int =
    if (n < 0) -n
    else n


  private def formatAbs(x: Int) = {
    val msg = "The absolute value of %d is %d"
    msg.format(x, abs(x))
  }


  def main(args: Array[String]): Unit =
    println(formatAbs(-42))
}
```

Declares a singleton object, which simultaneously declares a class and its only instance.

abs takes an integer and returns an integer.

Returns the negation of n if it's less than zero.

A private method can only be called by other members of MyModule.

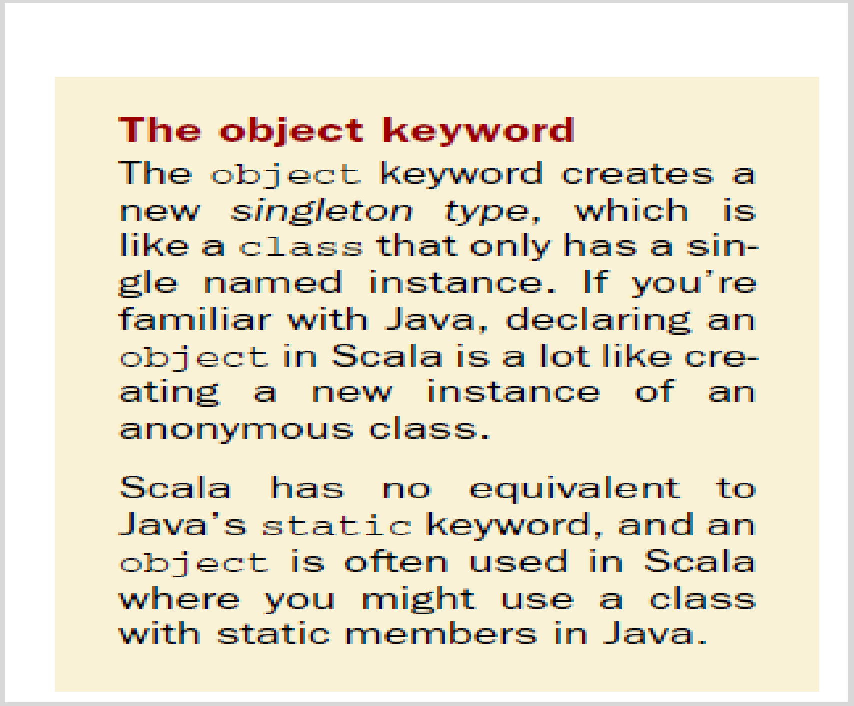
A string with two placeholders for numbers marked as %d.

Replaces the two %d placeholders in the string with x and abs(x) respectively.

Unit serves the same purpose as void in languages like Java or C.

Cntd..

- We declare an object (also known as a *module*) named `MyModule`. This is simply to give our code a place to live and a name so we can refer to it later. Scala code has to be in an `object` or a `class`, and we use an `object` here because it's simpler. We put our code inside the object, between curly braces.



The object keyword

The `object` keyword creates a new *singleton type*, which is like a `class` that only has a single named instance. If you're familiar with Java, declaring an `object` in Scala is a lot like creating a new instance of an anonymous class.

Scala has no equivalent to Java's `static` keyword, and an `object` is often used in Scala where you might use a class with static members in Java.

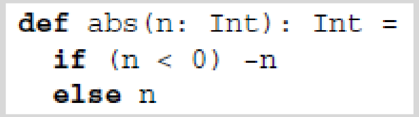
Singleton Object in Scala

◦ **Important points about singleton object**

- The method in the singleton object is globally accessible.
- You are not allowed to create an instance of singleton object.
- You are not allowed to pass parameter in the primary constructor of singleton object.
- In Scala, a singleton object can extend class and traits.
- In Scala, a main method is always present in singleton object.
- The method in the singleton object is accessed with the name of the object(just like calling static method in Java), so there is no need to create an object to access this method.

Cntd..

- The `abs` method is a pure function that takes an integer and returns its absolute value:



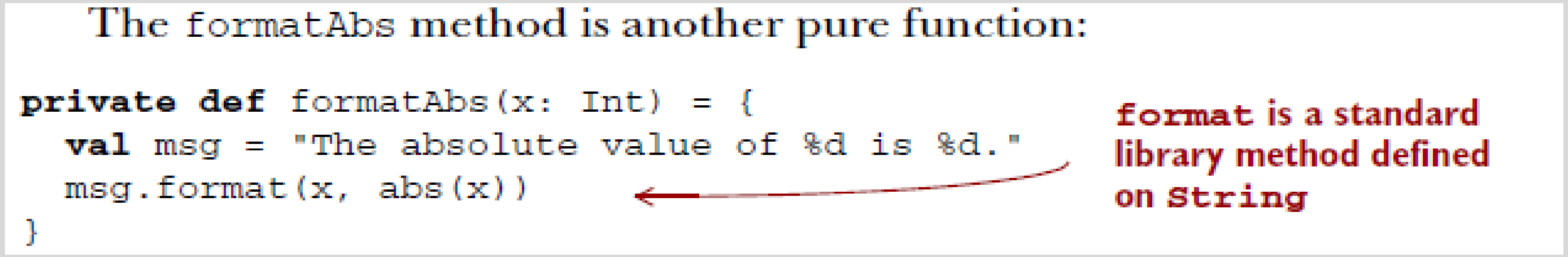
```
def abs(n: Int): Int =
  if (n < 0) -n
  else n
```

- The `def` keyword is followed by the name of the method, which is followed by the parameter list in parentheses. In this case, `abs` takes only one argument, `n` of type `Int`. Following the closing parenthesis of the argument list, an optional type annotation (the `: Int`) indicates that the type of the result is `Int` (the colon is pronounced “has type”). The body of the method itself comes after a single equals sign (=).

Cntd..

- We'll sometimes refer to the part of a declaration that goes before the equals sign as the *left-hand side* or *signature*, and the code that comes after the equals sign as the *right-hand side* or *definition*. Note the absence of an explicit `return` keyword. The value returned from a method is simply whatever value results from evaluating the right-hand side. All expressions, including `if` expressions, produce a result. Here, the right-hand side is a single expression whose value is either `-n` or `n`, depending on whether `n < 0`.

Cntd..



The formatAbs method is another pure function:

```
private def formatAbs(x: Int) = {
  val msg = "The absolute value of %d is %d."
  msg.format(x, abs(x))
}
```

format is a standard library method defined on String

Here we're calling the format method on the msg object, passing in the value of x along with the value of abs applied to x. This results in a new string with the occurrences of %d in msg replaced with the evaluated results of x and abs(x), respectively. This method is declared private, which means that it can't be called from any code outside of the MyModule object. This function takes an Int and returns a String, but note that the return type is not declared.

- Scala is usually able to infer the return types of methods, so they can be omitted, but it’s generally considered good style to explicitly declare the return types of methods that you expect others to use. This method is private to our module, so we can omit the type annotation.
- Remember, a method simply returns the value of its right-hand side, so we don’t need a `return` keyword. In this case, the right-hand side is a block. In Scala, the value of a multistatement block inside curly braces is the same as the value returned by the last expression in the block. Therefore, the result of the `formatAbs` method is just the value returned by the call to `msg.format(x, abs(x))`.

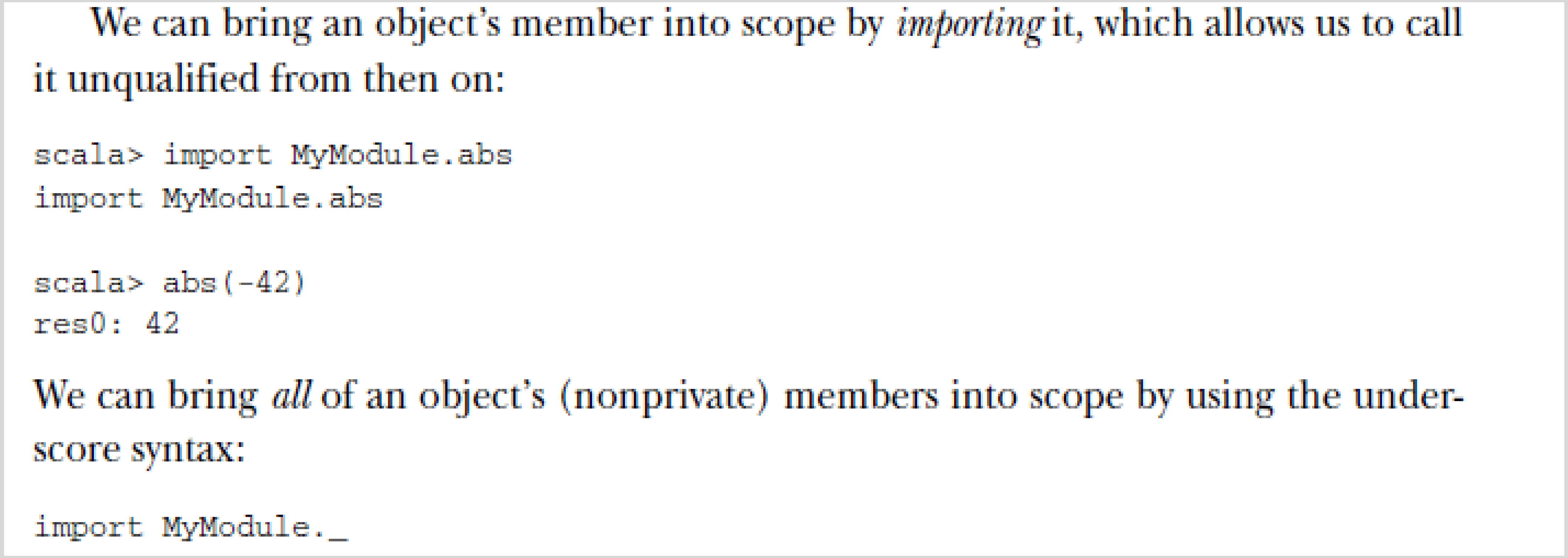
- Finally, our main method is an outer shell that calls into our purely functional core and prints the answer to the console. We'll sometimes call such methods procedures (or impure functions) rather tha functions, to emphasize the fact that they have side effects:
- def main(args: Array[String]): Unit =

      println(formatAbs(-42))
- `Unit` serves a similar purpose to `void` in programming languages like C and Java.
- In Scala, every method has to return some value as long as it doesn't crash or hang.

Modules, objects, and namespaces

- we’ll discuss some additional aspects of Scala’s syntax related to modules, objects, and namespaces.
- In the preceding session, note that in order to refer to our abs method, we had to say MyModule.abs because abs was defined in the MyModule object. We say that MyModule is its namespace. Aside from some technicalities, every value in Scala is what’s called an object and each object may have zero or more members. An object whose primary purpose is giving its members a namespace is sometimes called a module. A member can be a method declared with the def keyword, or it can be another object declared with val or object. Objects can also have other kinds of members that we’ll ignore for now.

- Note that even an expression like 2 + 1 is just calling a member of an object. In that case, what we're calling is the + member of the object 2. It's really syntactic sugar for the expression 2.+(1), which passes 1 as an argument to the method + on the object 2.
- Scala has no special notion of operators. It's simply the case that + is a valid method name in Scala. Any method name can be used infix like that (omitting the dot and parentheses) when calling it with a single argument. For example, instead of MyModule.abs(42) we can say MyModule abs 42 and get the same result.
- You can use whichever you find more pleasing in any given case.



We can bring an object's member into scope by *importing* it, which allows us to call it unqualified from then on:

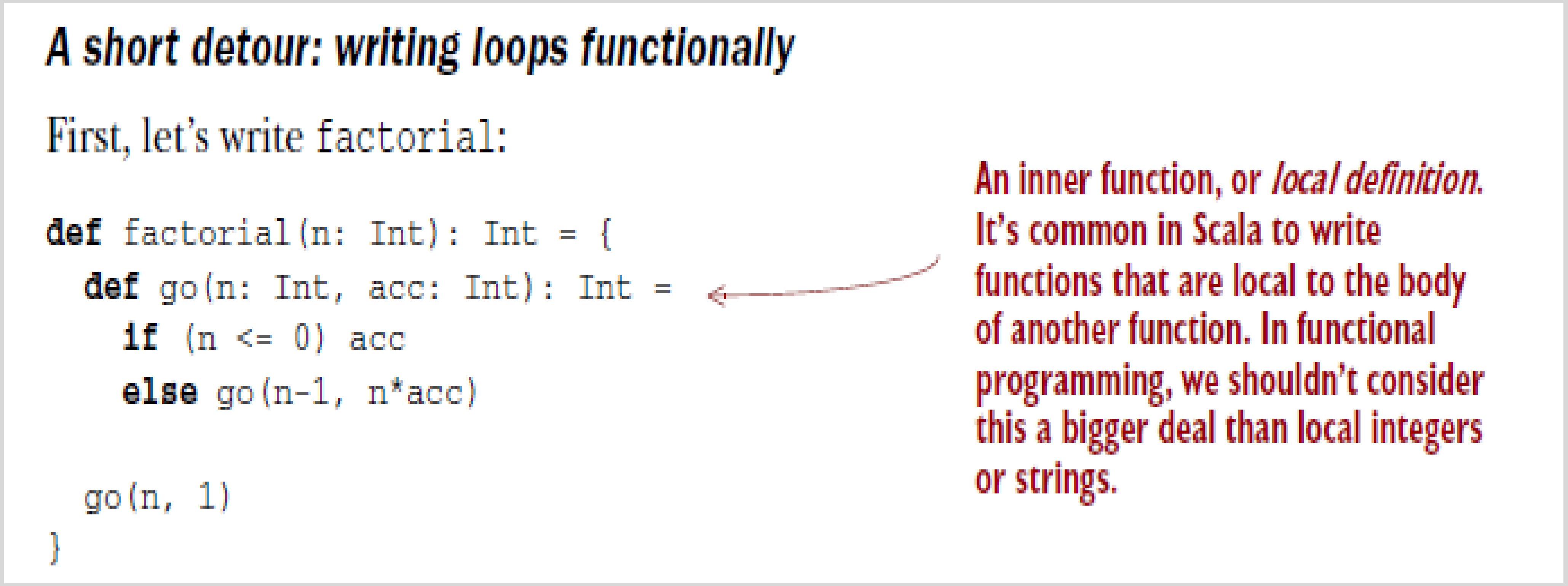
```
scala> import MyModule.abs
import MyModule.abs

scala> abs(-42)
res0: 42
```

We can bring *all* of an object's (nonprivate) members into scope by using the underscore syntax:

```
import MyModule._
```

Higher-order functions: passing functions to functions

- Now that we've covered the basics of Scala's syntax, we'll move on to covering some of the basics of writing functional programs. The first new idea is this: functions are values. And just like values of other types—such as integers, strings, and lists—functions can be assigned to variables, stored in data structures, and passed as arguments to functions.
- When writing purely functional programs, we'll often find it useful to write a function that accepts other functions as arguments. This is called a *higher-order function (HOF)*, and we'll look next at some simple examples to illustrate. In later lectures, we'll see how useful this capability really is, and how it permeates the functional programming style. But to start, suppose we wanted to adapt our program to print out both the absolute value of a number *and* the factorial of another number. Here's a
- sample run of such a program:
- `The absolute value of -42 is 42`
- `The factorial of 7 is 5040`



A short detour: writing loops functionally

First, let's write factorial:

```scala
def factorial(n: Int): Int = {
  def go(n: Int, acc: Int): Int =
    if (n <= 0) acc
    else go(n-1, n*acc)

  go(n, 1)
}
```

An inner function, or *local definition*. It's common in Scala to write functions that are local to the body of another function. In functional programming, we shouldn't consider this a bigger deal than local integers or strings.

- The way we write loops functionally, without mutating a loop variable, is with a recursive function. Here we’re defining a recursive helper function inside the body of the factorial function. Such a helper function is often called go or loop by convention.

- In Scala, we can define functions inside any block, including within another function definition. Since it’s local, the go function can only be referred to from within the body of the factorial function, just like a local variable would. The definition of factorial finally just consists of a call to go with the initial conditions for the loop.

- In Scala, we can define functions inside any block, including within another function definition. Since it's local, the go function can only be referred to from within the body of the factorial function, just like a local variable would. The definition of factorial finally just consists of a call to go with the initial conditions for the loop.

- The arguments to `go` are the state for the loop. In this case, they're the remaining value `n`, and the current accumulated factorial `acc`. To advance to the next iteration, we simply call `go` recursively with the new loop state (here, `go(n-1, n*acc)`), and to exit from the loop, we return a value without a recursive call (here, we return `acc` in the case that `n <= 0`). Scala detects this sort of *self-recursion* and compiles it to the same sort of bytecode as would be emitted for a `while` loop,3 so long as the recursive call is in *tail position*. See the sidebar for the technical details on this, but the basic idea is that this optimization4 (called *tail call elimination*) is applied when there's no additional work left to do after the recursive call returns.

- compiler about this assumption using the tailrec annotation , so it can give us a compile error if it’s unable to eliminate the tail calls of the function. Here’s the syntax for this:

```
◦ def factorial(n: Int): Int = {
◦       @annotation.tailrec
◦       def go(n: Int, acc: Int): Int =
◦              if (n <= 0) acc
◦             else go(n-1, n*acc)
◦        go(n, 1)
◦ }
```

- We won’t talk much more about annotations in these lectures, but we’ll use @annotation.tailrec extensively.

Exercise-1

- Write a recursive function to get the nth Fibonacci number. The first two Fibonacci numbers are 0 and 1. The nth number is always the sum of the previous two—the sequence begins 0, 1, 1, 2, 3, 5. Your definition should use local tail-recursive function.
- def fib(n: Int): Int

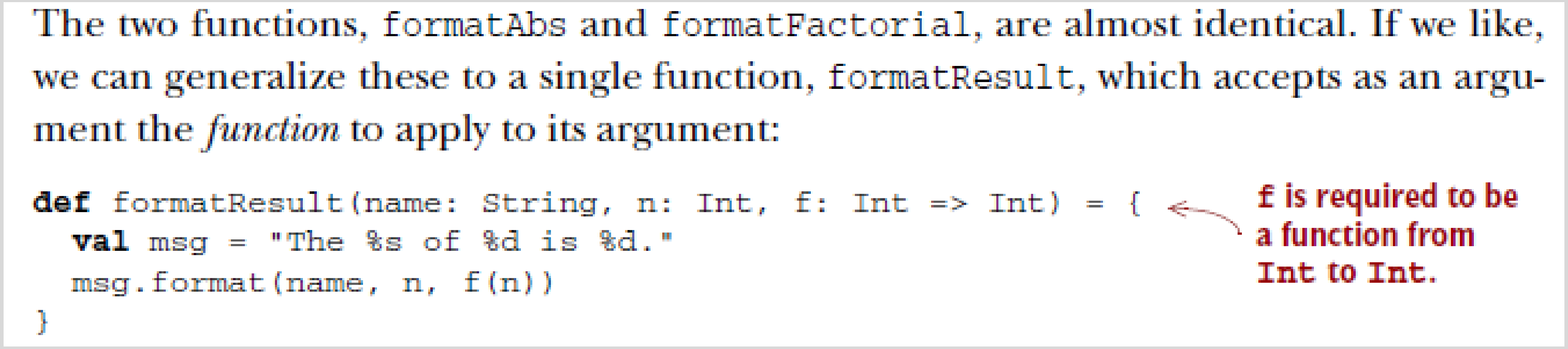
Writing our first higher-order function

Now that we have `factorial`, let's edit our program from before to include it.

Listing 2.2 A simple program including the factorial function

```
object MyModule {
  ...
  private def formatAbs(x: Int) = {
    val msg = "The absolute value of %d is %d."
    msg.format(x, abs(x))
  }

  private def formatFactorial(n: Int) = {
    val msg = "The factorial of %d is %d."
    msg.format(n, factorial(n))
  }

  def main(args: Array[String]): Unit = {
    println(formatAbs(-42))
    println(formatFactorial(7))
  }
}
```

Definitions of `abs` and `factorial` go here.



The two functions, formatAbs and formatFactorial, are almost identical. If we like, we can generalize these to a single function, formatResult, which accepts as an argument the *function* to apply to its argument:

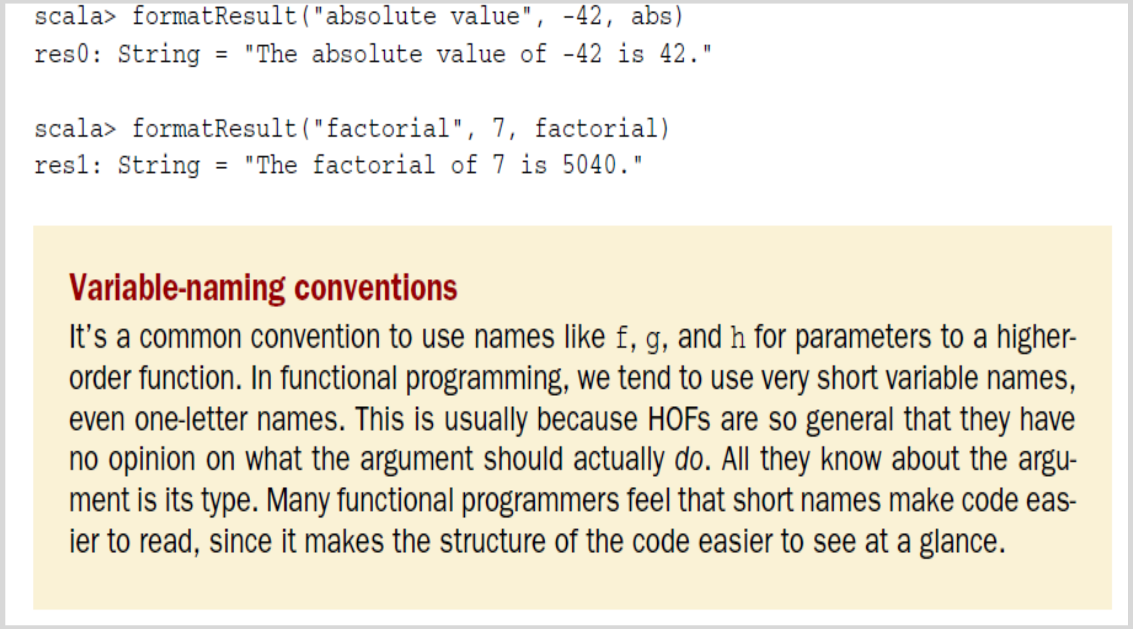
```
def formatResult(name: String, n: Int, f: Int => Int) = {
  val msg = "The %s of %d is %d."
  msg.format(name, n, f(n))
}
```

f is required to be a function from Int to Int.

HoF

- Our formatResult function is a higher-order function (HOF) that takes another function, called f (see sidebar on variable-naming conventions). We give a type to f, as we would for any other parameter. Its type is Int => Int (pronounced "int to int" or "int arrow int"), which indicates that f expects an integer argument and will also return an integer.

- Our function abs from before matches that type. It accepts an Int and returns an Int. And likewise, factorial accepts an Int and returns an Int, which also matches the Int => Int type. We can therefore pass abs or factorial as the f argument to
- formatResult:



```
scala> formatResult("absolute value", -42, abs)
res0: String = "The absolute value of -42 is 42."

scala> formatResult("factorial", 7, factorial)
res1: String = "The factorial of 7 is 5040."
```

Variable-naming conventions

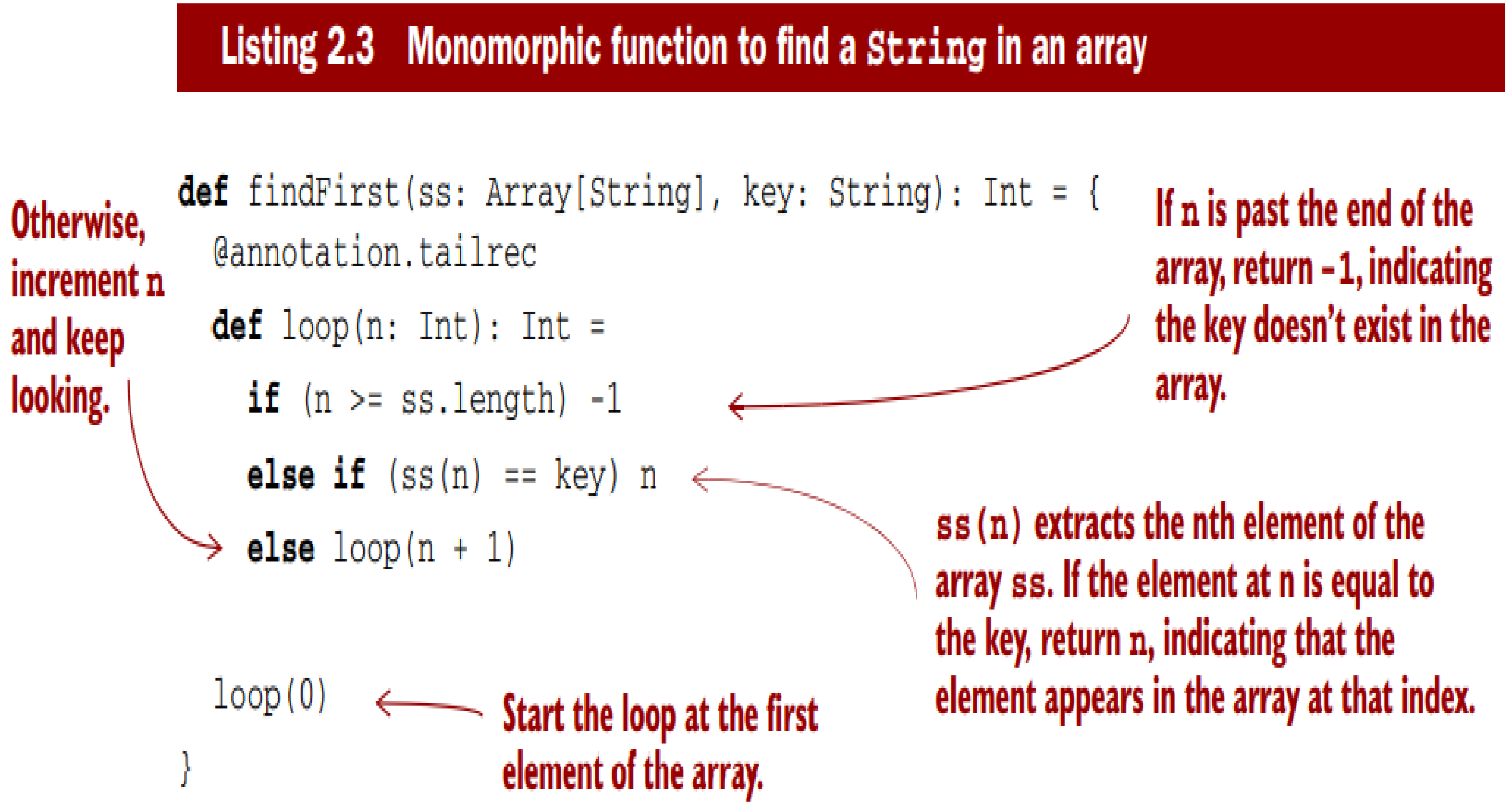
It's a common convention to use names like `f`, `g`, and `h` for parameters to a higher-order function. In functional programming, we tend to use very short variable names, even one-letter names. This is usually because HOFs are so general that they have no opinion on what the argument should actually *do*. All they know about the argument is its type. Many functional programmers feel that short names make code easier to read, since it makes the structure of the code easier to see at a glance.

Polymorphic functions: abstracting over types

To Be Continued…

Suggest part-2

Saturday or Sunday



Listing 2.3 Monomorphic function to find a String in an array

```scala
def findFirst(ss: Array[String], key: String): Int = {
  @annotation.tailrec
  def loop(n: Int): Int =
    if (n >= ss.length) -1
    else if (ss(n) == key) n
    else loop(n + 1)

  loop(0)
}
```

Otherwise, increment n and keep looking.

If n is past the end of the array, return -1, indicating the key doesn't exist in the array.

ss(n) extracts the nth element of the array ss. If the element at n is equal to the key, return n, indicating that the element appears in the array at that index.

Start the loop at the first element of the array.

We can often discover polymorphic functions by observing that several monomorphic

functions all share a similar structure. For example, the following monomorphic function, findFirst, returns the first index in an array where the key occurs, or -1 if it's

not found. It's specialized for searching for a String in an Array of String values.

```scala
def findFirst[A](as: Array[A], p: A => Boolean): Int = {
  @annotation.tailrec
  def loop(n: Int): Int =
    if (n >= as.length) -1
    else if (p(as(n))) n
    else loop(n + 1)

  loop(0)
}
```

Instead of hardcoding `String`, take a type `A` as a parameter. And instead of hardcoding an equality check for a given key, take a function with which to test each element of the array.

If the function `p` matches the current element, we've found a match and we return its index in the array.

Polymorphic function to find an element in an array

The details of the code aren't too important here. What's important is that the code for findFirst will look almost identical if we're searching for a String in an Array[String], an Int in an Array[Int], or an A in an Array[A] for any given type A. We can write findFirst more generally for any type A by accepting a function to use for testing a particular A value.

Cntd.

- This is an example of a polymorphic function, sometimes called a generic function. We’re abstracting over the type of the array and the function used for searching it. To write a polymorphic function as a method, we introduce a comma-separated list of type parameters, surrounded by square brackets (here, just a single [A]), following the name of the function, in this case findFirst. We can call the type parameters anything we want—[Foo, Bar, Baz] and [TheParameter, another_good_one] are valid type parameter declarations—though by convention we typically use short, one-letter, uppercase type parameter names like [A,B,C].

Following types to implementations

- Let’s look at an example of a function signature that can only be implemented in one way. It’s a higher-order function for performing what’s called partial application. This function, partial1, takes a value and a function of two arguments, and returns a function of one argument as its result. The name comes from the fact that the function is being applied to some but not all of the arguments it requires: def partial1[A,B,C](a: A, f: (A,B) => C): B => C

Cntd.

- The partial1 function has three type parameters: A, B, and C. It then takes two arguments. The argument f is itself a function that takes two arguments of types A and B, respectively, and returns a value of type C. The value returned by partial1 will also be a function, of type B => C. How would we go about implementing this higher-order function? It turns out that there's only one implementation that compiles, and it follows logically from the type signature. It's like a fun little logic puzzle.7 Let's start by looking at the type of thing that we have to return. The return type of partial1 is B => C, so we know that we have to return a function of that type. We can just begin writing a function literal that takes an argument of type B:
- def partial1[A,B,C](a: A, f: (A,B) => C): B => C = (b: B) => ???

- This can be weird at first if you're not used to writing anonymous functions. Where did that B come from? Well, we've just written, "Return a function that takes a value b of type B." On the right-hand-side of the => arrow (where the question marks are now) comes the body of that anonymous function. We're free to refer to the value b in there for the same reason that we're allowed to refer to the value a in the body of partial[8] Let's keep going. Now that we've asked for a value of type B, what do we want to return from our anonymous function? The type signature says that it has to be a value of type C. And there's only one way to get such a value. According to the signature, C is the return type of the function f. So the only way to get that C is to pass an A and a B to f. That's easy:
- `def partial1[A,B,C](a: A, f: (A,B) => C): B => C = (b: B) => f(a, b)`

- 8. Within the body of this inner function, the outer a is still in scope. We sometimes say that the inner function closes over its environment, which includes a.

- And we’re done! The result is a higher-order function that takes a function of two arguments and partially applies it. That is, if we have an A and a function that needs both A and B to produce C, we can get a function that just needs B to produce C (since we already have the A). It’s like saying, “If I can give you a carrot for an apple and a banana, and you already gave me an apple, you just have to give me a banana and I’ll give you a carrot.” Note that the type annotation on b isn’t needed here. Since we told Scala the return type would be B => C, Scala knows the type of b from the context and we could just write b => f(a,b) as the implementation. Generally speaking, we’ll omit the type annotation on a function literal if it can be inferred by Scala

Example Currying

- Let’s look at another example, currying, which converts a function f of two arguments into a function of one argument that partially applies f. Here again there’s only one implementation that compiles. Write this implementation.
- def curry[A,B,C](f: (A, B) => C): A => (B => C)

Example Uncrruying

- Implement uncurry, which reverses the transformation of curry. Note that since =>
- associates to the right, A => (B => C) can be written as A => B => C.
- def uncurry[A,B,C](f: A => B => C): (A, B) => C

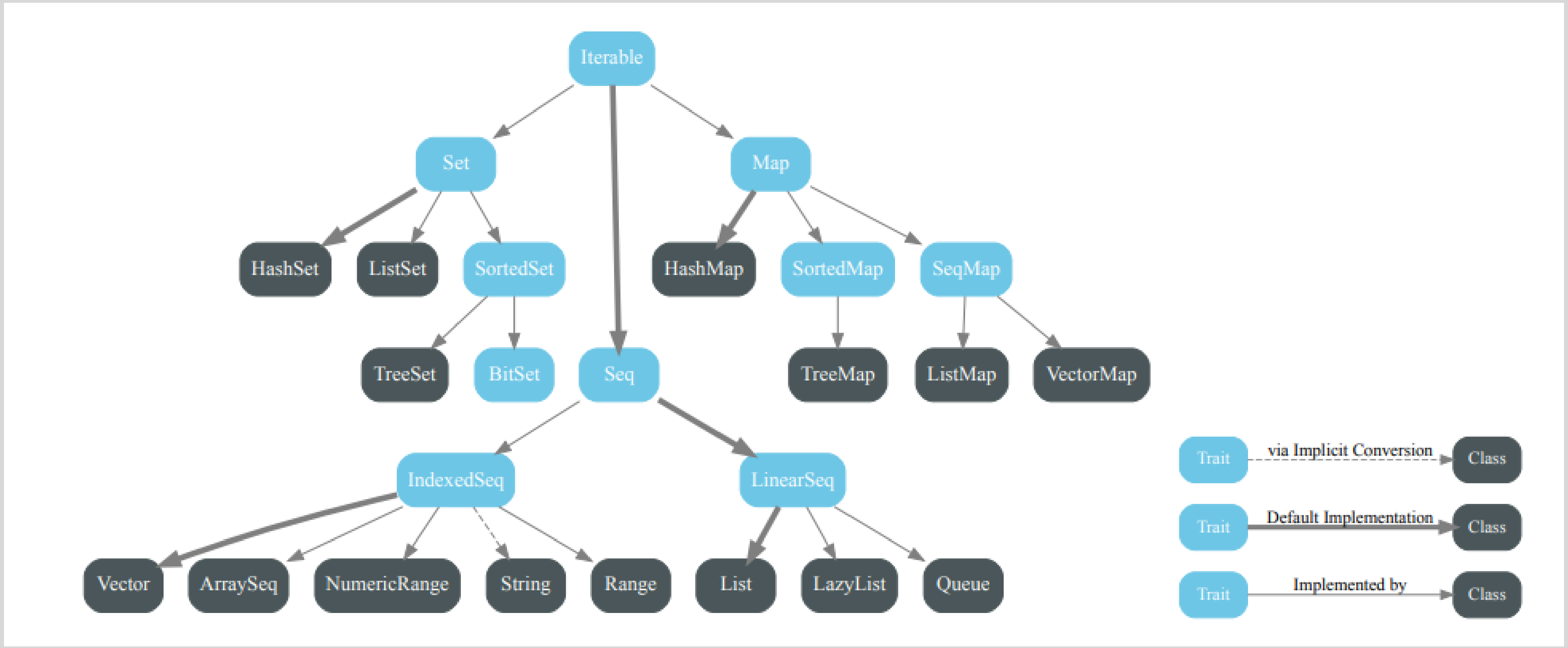
Map, List, tuple….

To Be Continued…

Suggested Tuesday!!

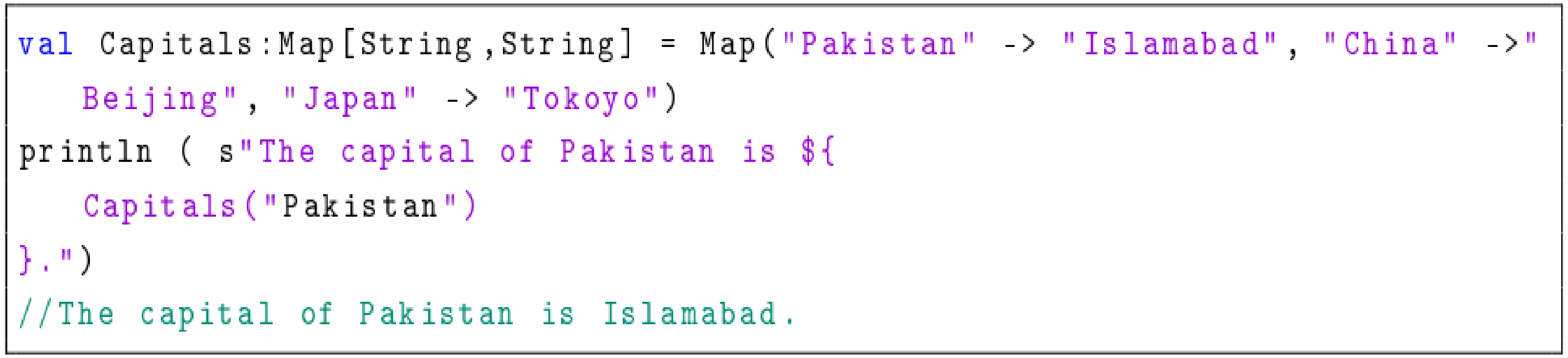
Scala Collections

- Scala has rich collections libraries. Collections serve as containers for data of similar or different types, which can be efficiently operated by the supported methods of the respective collection library. Scala collections are grouped in three packages or sub-packages. Using the keyword Seq implies immutable collection, while mutable.Seq will refer to mutable counterpart. Figure 10.11 shows the collections hierarchy for immutable collections.
- scala.collection: Includes both mutable and immutable collections
- scala.collection.mutable: Includes mutable collections
- scala.collection.immutable: Includes immutable collections


Iterable
Set
Map
HashSet
ListSet
SortedSet
HashMap
SortedMap
SeqMap
TreeSet
BitSet
Seq
TreeMap
ListMap
VectorMap
IndexedSeq
LinearSeq
Vector
ArraySeq
NumericRange
String
Range
List
LazyList
Queue
Trait
via Implicit Conversion
Class
Trait
Default Implementation
Class
Trait
Implemented by
Class

Self Study Topics in Scala

- Array
- List
- Tuple
- Set



```
val Capitals:Map[String,String] = Map("Pakistan" -> "Islamabad", "China" ->"
   Beijing", "Japan" -> "Tokoyo")
println ( s"The capital of Pakistan is ${
   Capitals("Pakistan")
}.")
//The capital of Pakistan is Islamabad.
```

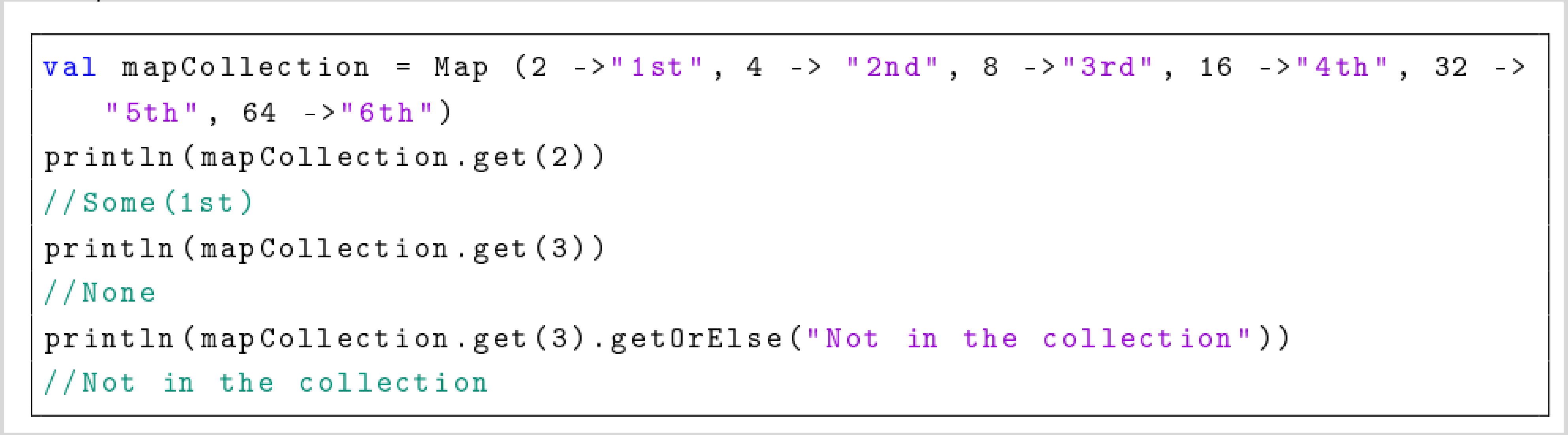
Map in Scala

The Map collection works as a key-value pair. Keys in a Map are always unique, so keys are actually a set, in which a value is associated with each member of the set. Values can be duplicate. Keys and values can have different data types.

Both Set and Map collections have mutable and immutable flavors. Here we are using a default type of Map, which is immutable

Options

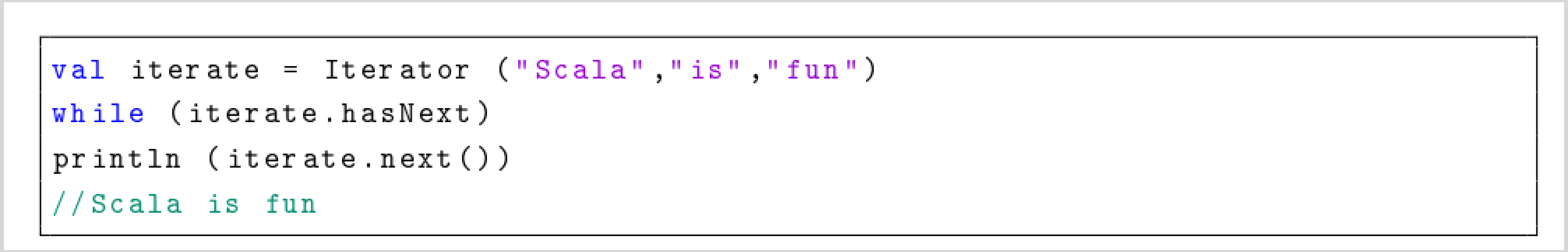
- Sometime we are not sure if a collection contains a certain element or not. In such cases we can use option type. Option type in Scala is referred to as a carrier having one or no element of the stated type. We can check the presence of a specific element using the get method. For instance, the get method for the collection returns Some if the value is found, otherwise it returns None. Options can also be used in other situations. For example, it can be used as a function parameter. Figure illustrate using get method on Map.



```
val mapCollection = Map (2 ->"1st", 4 -> "2nd", 8 ->"3rd", 16 ->"4th", 32 ->
   "5th", 64 ->"6th")
println(mapCollection.get(2))
//Some(1st)
println(mapCollection.get(3))
//None
println(mapCollection.get(3).getOrElse("Not in the collection"))
//Not in the collection
```

Iterator

- Iterator is a collection that works similar to a queue. We can get the head by calling next method on it. To check if the iterator is empty we call hasnext method on it. Method hasnext returns a Boolean value to tell if iterator is empty or not.
- You can only go through an Iterator once because it is consumed as you go through it. The reason for using an Iterator is generally for performance and memory benefits. There are methods that return an Iterator, for example, the .combinations method returns an Iterator.



```
val iterate = Iterator ("Scala","is","fun")
while (iterate.hasNext)
println (iterate.next())
//Scala is fun
```

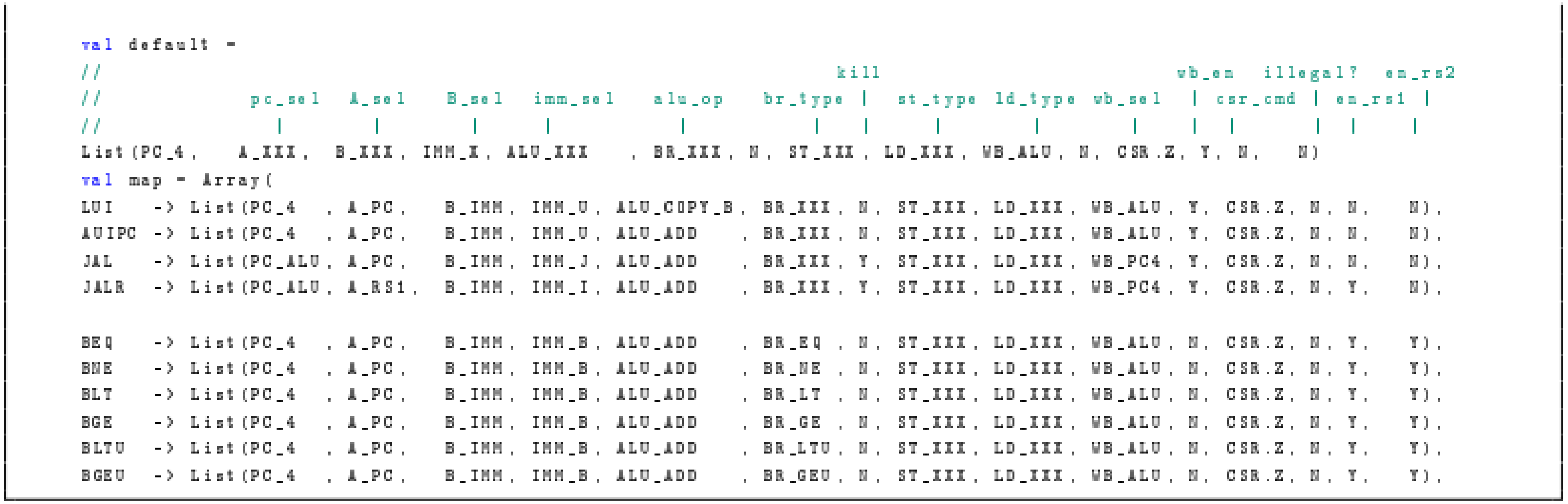
Controller Design

- In this section we will look into the controller implementation for a RISC V microprocessor, which takes advantage of different Scala collections. Controller implementation is one of the most critical steps in the processor realization. In Figure. different control signals, including inputs to the controller and outputs from the controller, have been defined as a concrete class.

```
class ControlSignals extends Bundle with Config {
   val inst      = Input(UInt(XLEN.W))
   val pc_sel    = Output(UInt(2.W))
   val inst_kill = Output(Bool())
   val A_sel     = Output(UInt(1.W))
   val B_sel     = Output(UInt(1.W))
   val imm_sel   = Output(UInt(3.W))
   val alu_op    = Output(UInt(5.W))
   val br_type   = Output(UInt(3.W))
   val st_type   = Output(UInt(2.W))
   val ld_type   = Output(UInt(3.W))
   val wb_sel    = Output(UInt(2.W))
   val wb_en     = Output(Bool())
   val csr_cmd   = Output(UInt(3.W))
   val illegal   = Output(Bool())
   val en_rs1    = Output(Bool())
   val en_rs2    = Output(Bool())
}
```

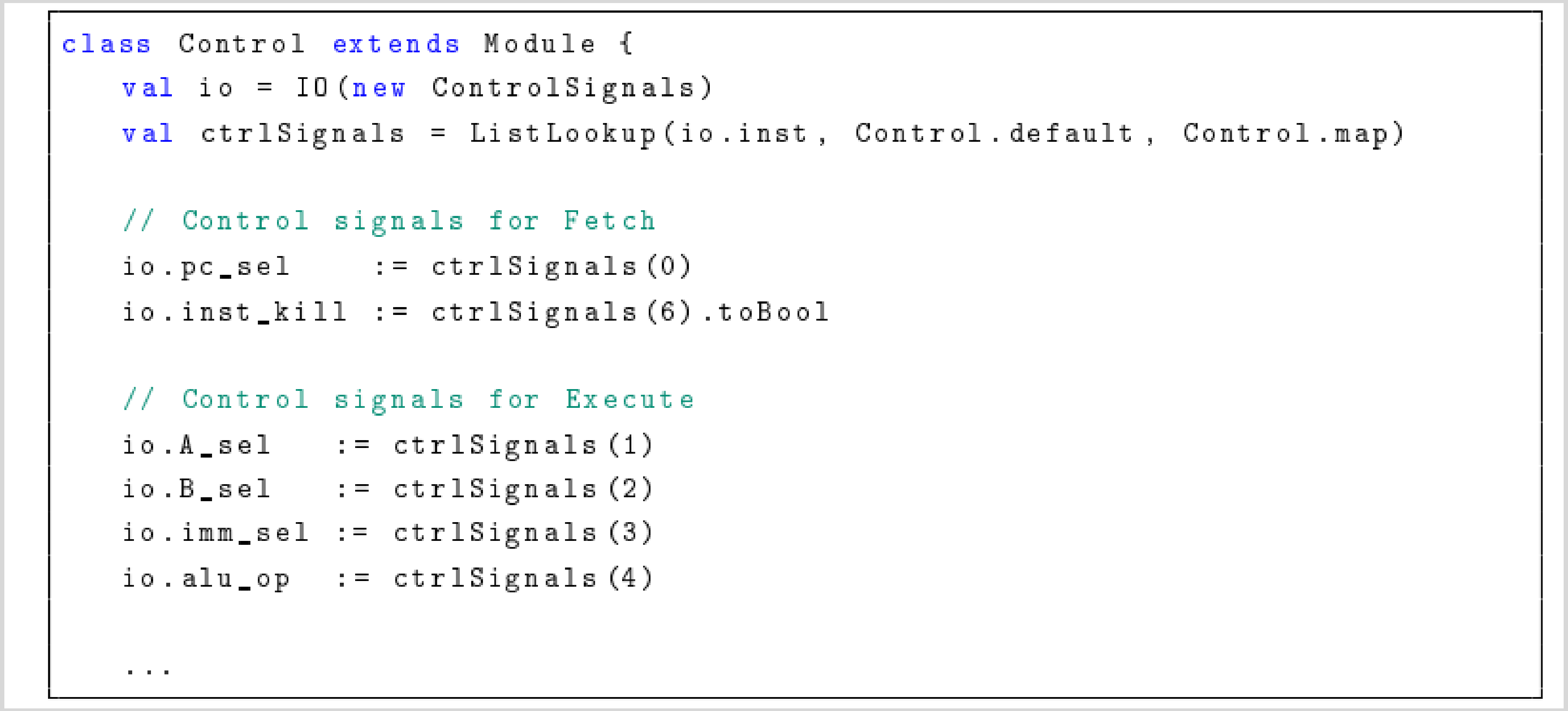
- The input to the controller is a RISC V instruction, which is looked up from the control table and corresponding control signals are generated. The control signals are used by different modules of the processor to perform their function in the context of respective instruction. The control table with partial entries is provided in below figure . In other figure, a Controller is implemented using ListLookup, which is a Chisel construct.

Control signals for different instructions



```
val default =
//                                                    kill                       wb_en  illegal?  en_rs2
//          pc_sel  A_sel   B_sel   imm_sel   alu_op     br_type |  st_type ld_type wb_sel  | csr_cmd | en_rs1 |
//             |       |       |       |          |           |  |     |        |      |    |  |      |   |    |
List(PC_4 ,   A_XXX,  B_XXX, IMM_X, ALU_XXX    , BR_XXX, N, ST_XXX, LD_XXX, WB_ALU, N, CSR.Z, Y, N,   N)
val map = Array(
LUI   -> List(PC_4  , A_PC,   B_IMM, IMM_U, ALU_COPY_B, BR_XXX, N, ST_XXX, LD_XXX, WB_ALU, Y, CSR.Z, N, N,   N),
AUIPC -> List(PC_4  , A_PC,   B_IMM, IMM_U, ALU_ADD   , BR_XXX, N, ST_XXX, LD_XXX, WB_ALU, Y, CSR.Z, N, N,   N),
JAL   -> List(PC_ALU, A_PC,   B_IMM, IMM_J, ALU_ADD   , BR_XXX, Y, ST_XXX, LD_XXX, WB_PC4, Y, CSR.Z, N, N,   N),
JALR  -> List(PC_ALU, A_RS1,  B_IMM, IMM_I, ALU_ADD   , BR_XXX, Y, ST_XXX, LD_XXX, WB_PC4, Y, CSR.Z, N, Y,   N),

BEQ   -> List(PC_4  , A_PC,   B_IMM, IMM_B, ALU_ADD   , BR_EQ , N, ST_XXX, LD_XXX, WB_ALU, N, CSR.Z, N, Y,   Y),
BNE   -> List(PC_4  , A_PC,   B_IMM, IMM_B, ALU_ADD   , BR_NE , N, ST_XXX, LD_XXX, WB_ALU, N, CSR.Z, N, Y,   Y),
BLT   -> List(PC_4  , A_PC,   B_IMM, IMM_B, ALU_ADD   , BR_LT , N, ST_XXX, LD_XXX, WB_ALU, N, CSR.Z, N, Y,   Y),
BGE   -> List(PC_4  , A_PC,   B_IMM, IMM_B, ALU_ADD   , BR_GE , N, ST_XXX, LD_XXX, WB_ALU, N, CSR.Z, N, Y,   Y),
BLTU  -> List(PC_4  , A_PC,   B_IMM, IMM_B, ALU_ADD   , BR_LTU, N, ST_XXX, LD_XXX, WB_ALU, N, CSR.Z, N, Y,   Y),
BGEU  -> List(PC_4  , A_PC,   B_IMM, IMM_B, ALU_ADD   , BR_GEU, N, ST_XXX, LD_XXX, WB_ALU, N, CSR.Z, N, Y,   Y),
```

Controller implementation using ListLookup



```
class Control extends Module {
   val io = IO(new ControlSignals)
   val ctrlSignals = ListLookup(io.inst, Control.default, Control.map)

   // Control signals for Fetch
   io.pc_sel    := ctrlSignals(0)
   io.inst_kill := ctrlSignals(6).toBool

   // Control signals for Execute
   io.A_sel   := ctrlSignals(1)
   io.B_sel   := ctrlSignals(2)
   io.imm_sel := ctrlSignals(3)
   io.alu_op  := ctrlSignals(4)

   ...
```

◦ The ListLookUp is roughly equivalent to a switch() statement in software programming, or a casez statement in Verilog RTL.

◦ The first argument (io.dat.inst) is the signal to match against.

◦ The second argument is the default value if no match occurs.

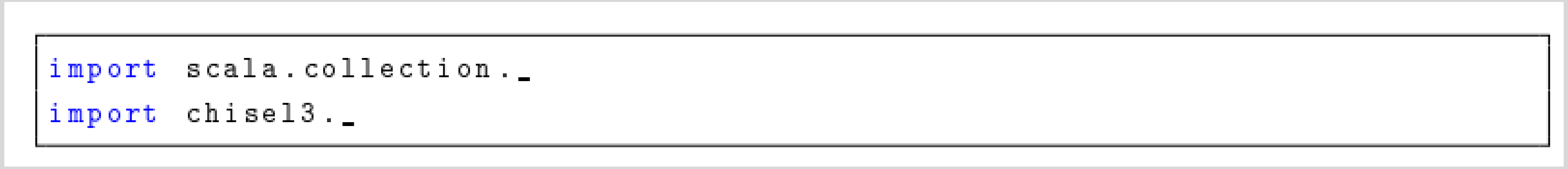
◦ The third argument is an Array of "key"->"value" tuples. The io.dat.inst is matched against the keys to find a match and the csignals is set to the value part if a match is found.

◦ So Cntrlsignals is of a type List().

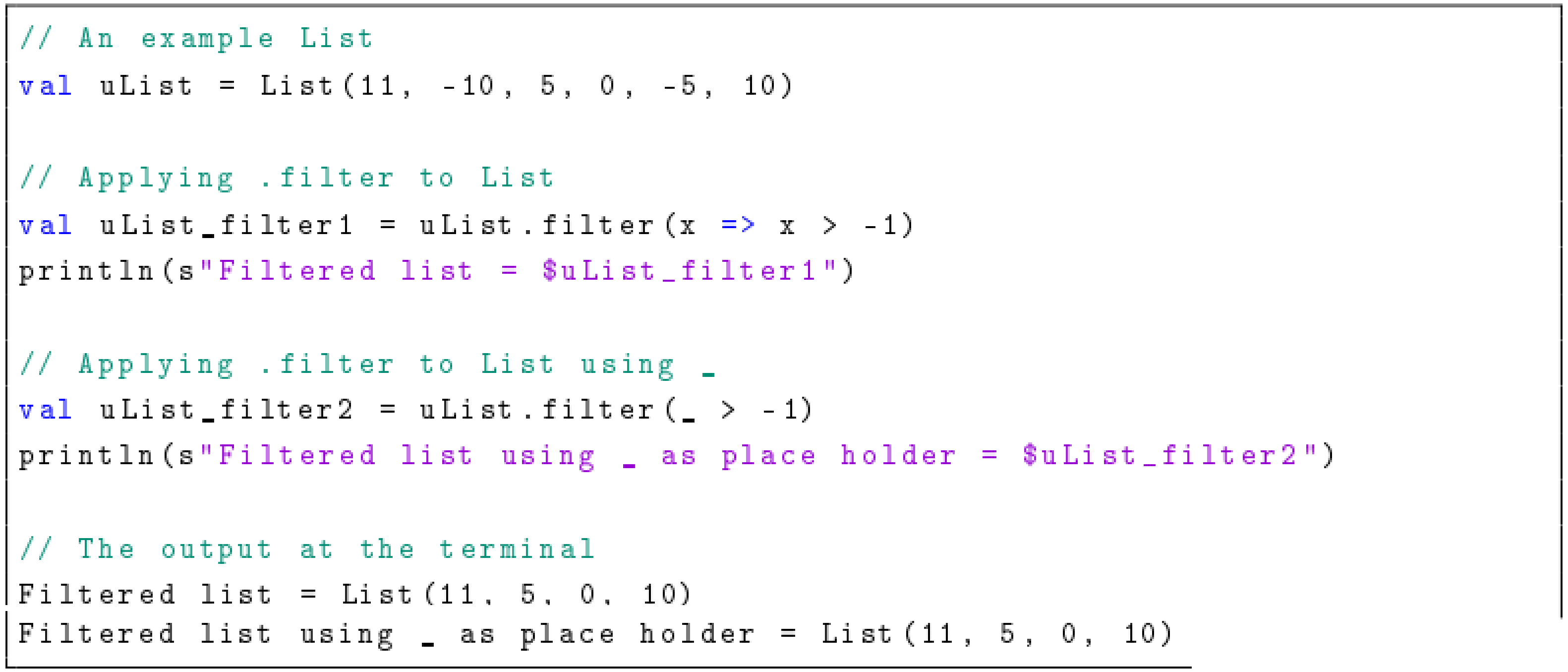
Scala Wildcard ‘_’

◦ A wildcard is a kind of placeholder represented by a single character, which happens to be ‘_ ’ in Scala. This attribute is also present in other programming languages, e.g. in Java, we use ‘*’ for wildcard. It will be illustrated in later sections that how a wildcard can also be used as a default case handler, to access a tuple selectively as well as to import classes or functions, to name a few. For instance, figure shows how a wildcard can be used to import necessary packages and libraries.

◦ Importing dependencies



```
import scala.collection._
import chisel3._
```

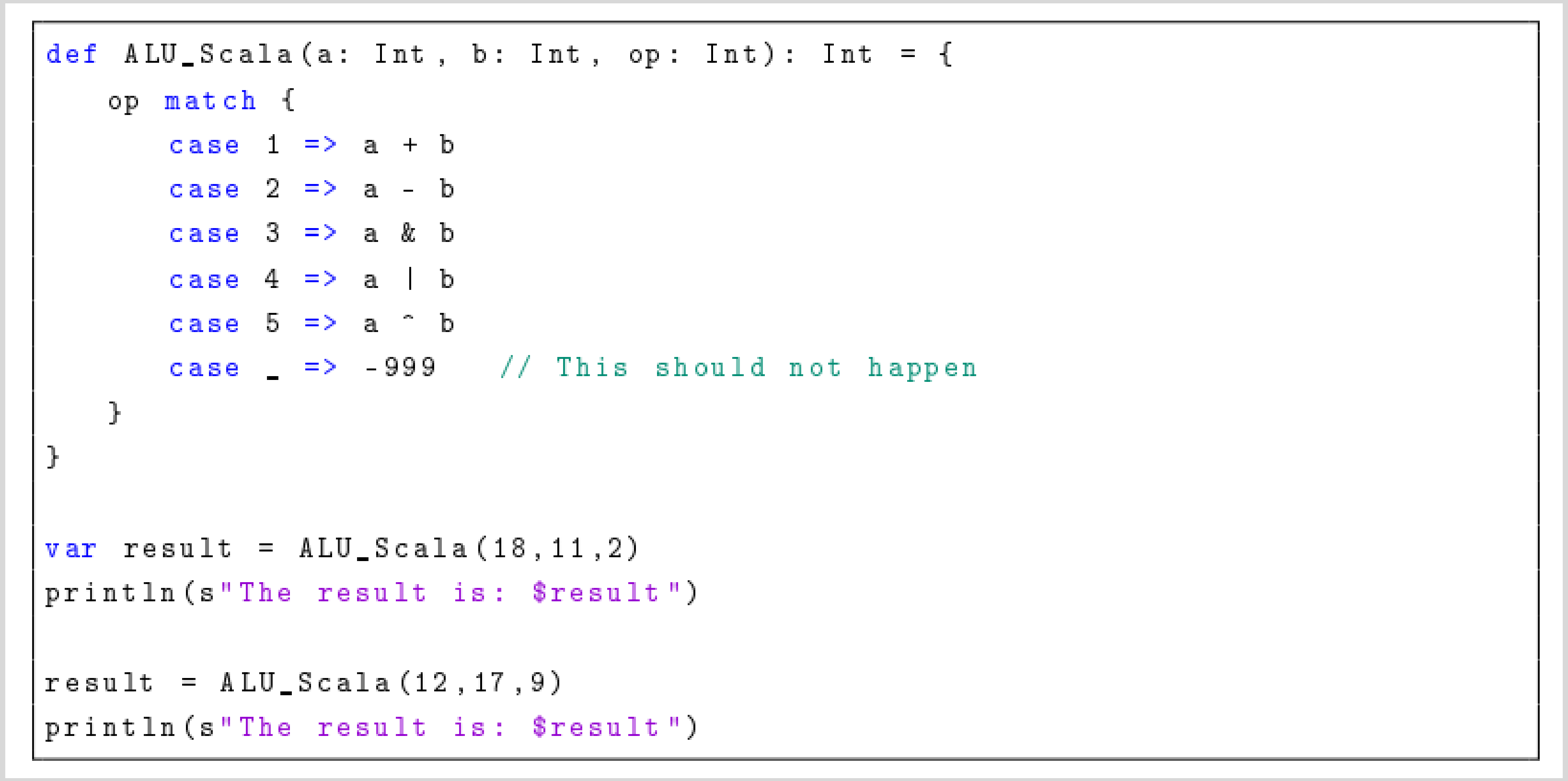


```scala
// An example List
val uList = List(11, -10, 5, 0, -5, 10)

// Applying .filter to List
val uList_filter1 = uList.filter(x => x > -1)
println(s"Filtered list = $uList_filter1")

// Applying .filter to List using _
val uList_filter2 = uList.filter(_ > -1)
println(s"Filtered list using _ as place holder = $uList_filter2")

// The output at the terminal
Filtered list = List(11, 5, 0, 10)
Filtered list using _ as place holder = List(11, 5, 0, 10)
```

Wildcard as Placeholder

Let us first see how a wildcard character can be used as a placeholder. In figure, a list is instantiated with both positive and negative integers. It is then modified using the filter method to remove or filter all negative integers. This is first done by calling filter method with explicit arguments. Next we perform the same filtering by using wildcard as placeholder for the arguments to the filter method. The output for both approaches will be the same as can be noted from figure.

Figure shows another use of wildcard in a Scala match statement. Here, the user defined method ALU Scala checks a list of possible cases against an input argument ‘op’ and then applies the respective operation on the input operands. Observe that for the last case, a wildcard character is used to define the default case. This is equivalent to the scenario, where the match function can not find a matching case against the ‘op’ argument supplied and as a result it selects the default case.



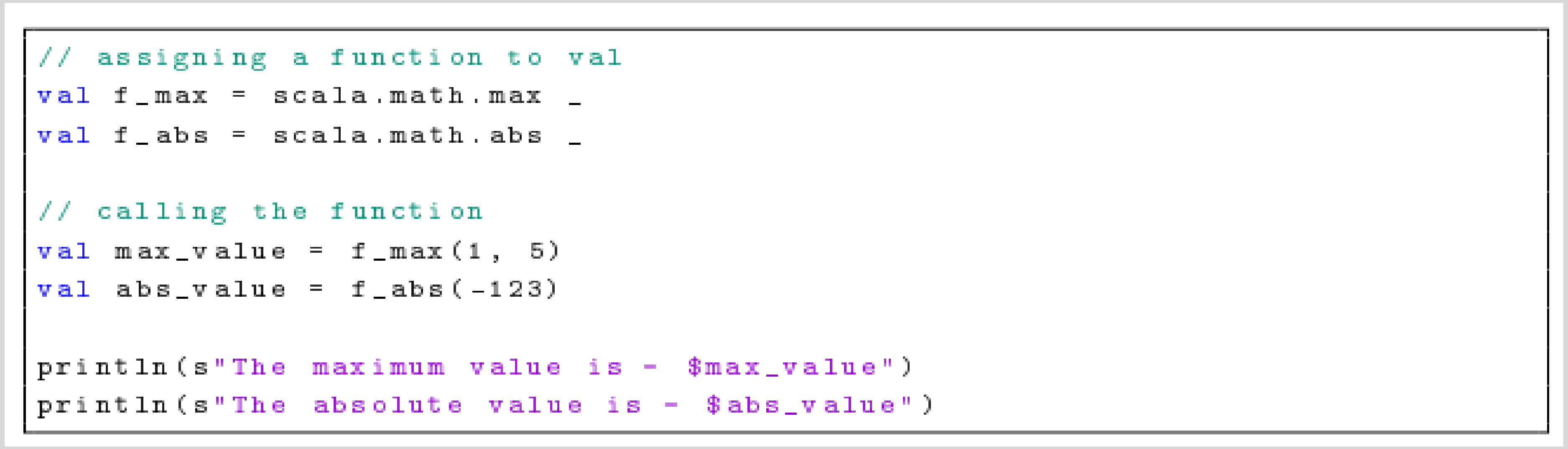
```
def ALU_Scala(a: Int, b: Int, op: Int): Int = {
    op match {
        case 1 => a + b
        case 2 => a - b
        case 3 => a & b
        case 4 => a | b
        case 5 => a ^ b
        case _ => -999   // This should not happen
    }
}

var result = ALU_Scala(18,11,2)
println(s"The result is: $result")

result = ALU_Scala(12,17,9)
println(s"The result is: $result")
```

```
// User type definition
type R = Int

// An example List
val uList = List(1, 2, 3, 4, 5)

// functional composition
def compose(g: R => R, h: R => R) =
(x: R) => g(h(x))

// implement y = mx+c ( with m=2 and c =1)
def y1 = compose(x => x + 1, x => x * 2)
def y2 = compose(_ + 1, _ * 2)

val uList_map1 = uList.map(x => y1(x))
val uList_map2 = uList.map(y2(_))

println(s" Linearly mapped list 1 = $uList_map1 ")
println(s" Linearly mapped list 2 = $uList_map2 ")
```

Similar to what we did with the filter method, we can define our own methods and use one or multiple wildcards, when calling these user defined methods. Figure illustrates this through an example. The user defined function 'compose' defines two integer type functional mappings as well as the order of the mappings. Functions y1 and y2 are defined using the 'compose' function and provide the expressions for the mapping functions. Both y1 and y2 are the same except that y2 uses wildcard to write the function expressions. Both functions are called on the same list i.e 'uList' and produce the same output.

Wildcard for Function Assignment

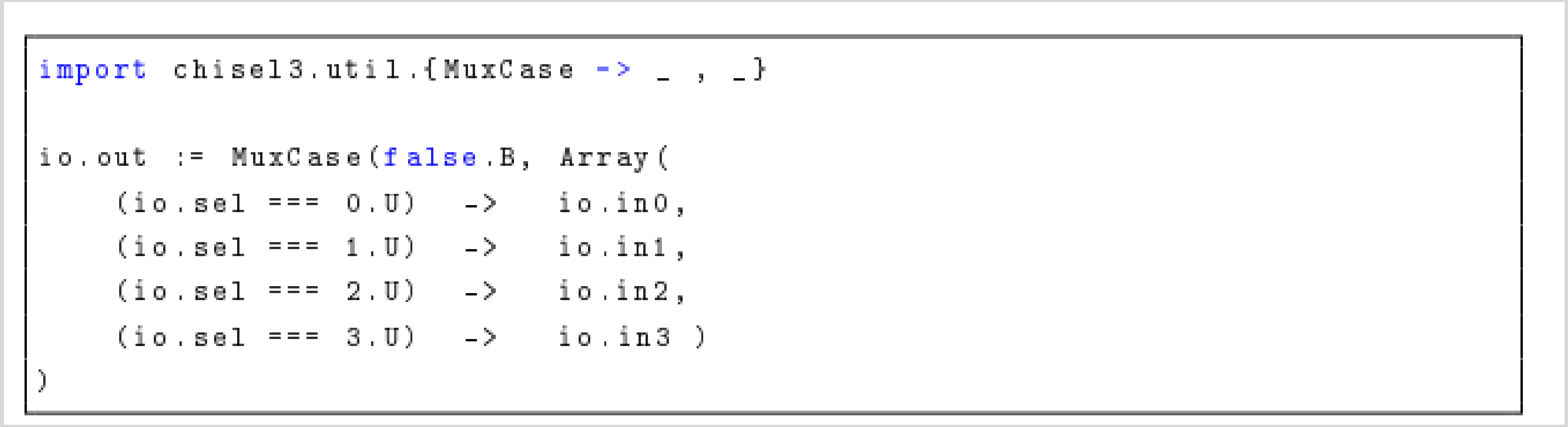
- Another possible use of wildcard is to assign methods, called from libraries, to variables as illustrated in figure Two methods max and abs are assigned to two variables and then instead of calling these two methods using long chained names, we can simply use variables as function callers. The required arguments to make function calls, are supplied the same way as one would do with the original function call



```scala
// assigning a function to val
val f_max = scala.math.max _
val f_abs = scala.math.abs _

// calling the function
val max_value = f_max(1, 5)
val abs_value = f_abs(-123)

println(s"The maximum value is - $max_value")
println(s"The absolute value is - $abs_value")
```

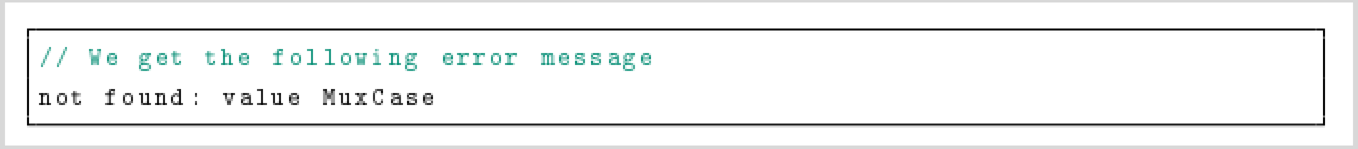
Hiding a Class During Import

- When importing a library, a specific class or object can be prevented from being imported by using wildcard as illustrated in figure. Figure 2 shows the error generated at the output terminal when we attempt to call the prevented class



```
import chisel3.util.{MuxCase -> _ , _}

io.out := MuxCase(false.B, Array(
    (io.sel === 0.U)  ->   io.in0,
    (io.sel === 1.U)  ->   io.in1,
    (io.sel === 2.U)  ->   io.in2,
    (io.sel === 3.U)  ->   io.in3 )
)
```

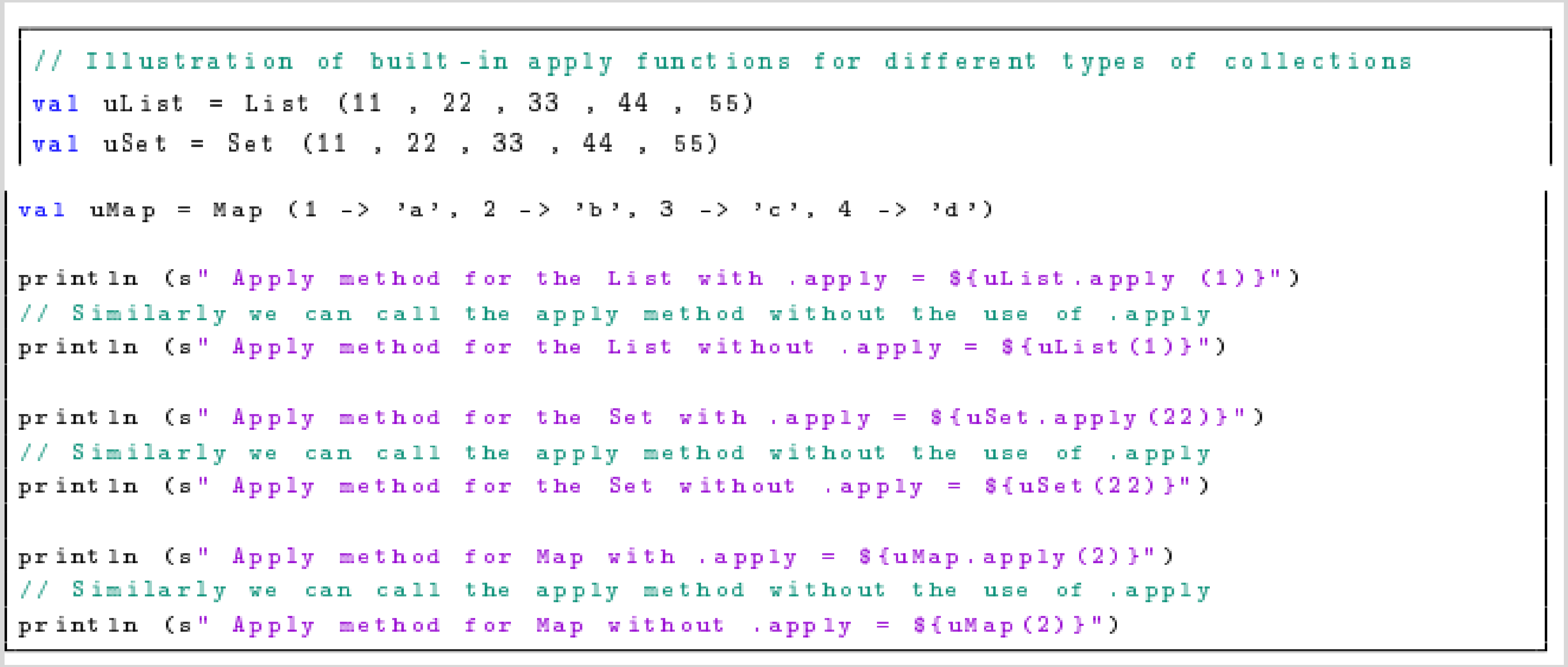
- Error



```
// We get the following error message
not found: value MuxCase
```

The apply method

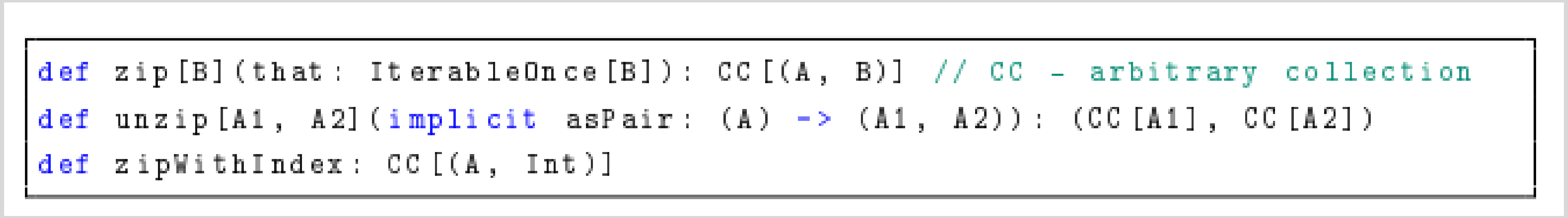
- The apply method is widely used as one of the default methods supported by Scala traits, classes and objects. For the three traits Seq, Set and Map the apply method functioning is summarized below.
- Seq: apply is positional indexing, element numbering starts from 0 always
- Set: apply is a membership test and is similar to the contains method
- Map: apply is a selection and returns the value associated with the given key Based on the above discussion, the use of apply method is illustrated in figure for three different collections, namely, List, Set and Map. Figure 10.1, showing the collections hierarchy for immutable collections, is reproduced here for quick reference.



```scala
// Illustration of built-in apply functions for different types of collections
val uList = List (11 , 22 , 33 , 44 , 55)
val uSet = Set (11 , 22 , 33 , 44 , 55)
val uMap = Map (1 -> 'a', 2 -> 'b', 3 -> 'c', 4 -> 'd')

println (s" Apply method for the List with .apply = ${uList.apply (1)}")
// Similarly we can call the apply method without the use of .apply
println (s" Apply method for the List without .apply = ${uList(1)}")

println (s" Apply method for the Set with .apply = ${uSet.apply(22)}")
// Similarly we can call the apply method without the use of .apply
println (s" Apply method for the Set without .apply = ${uSet(22)}")

println (s" Apply method for Map with .apply = ${uMap.apply(2)}")
// Similarly we can call the apply method without the use of .apply
println (s" Apply method for Map without .apply = ${uMap(2)}")
```

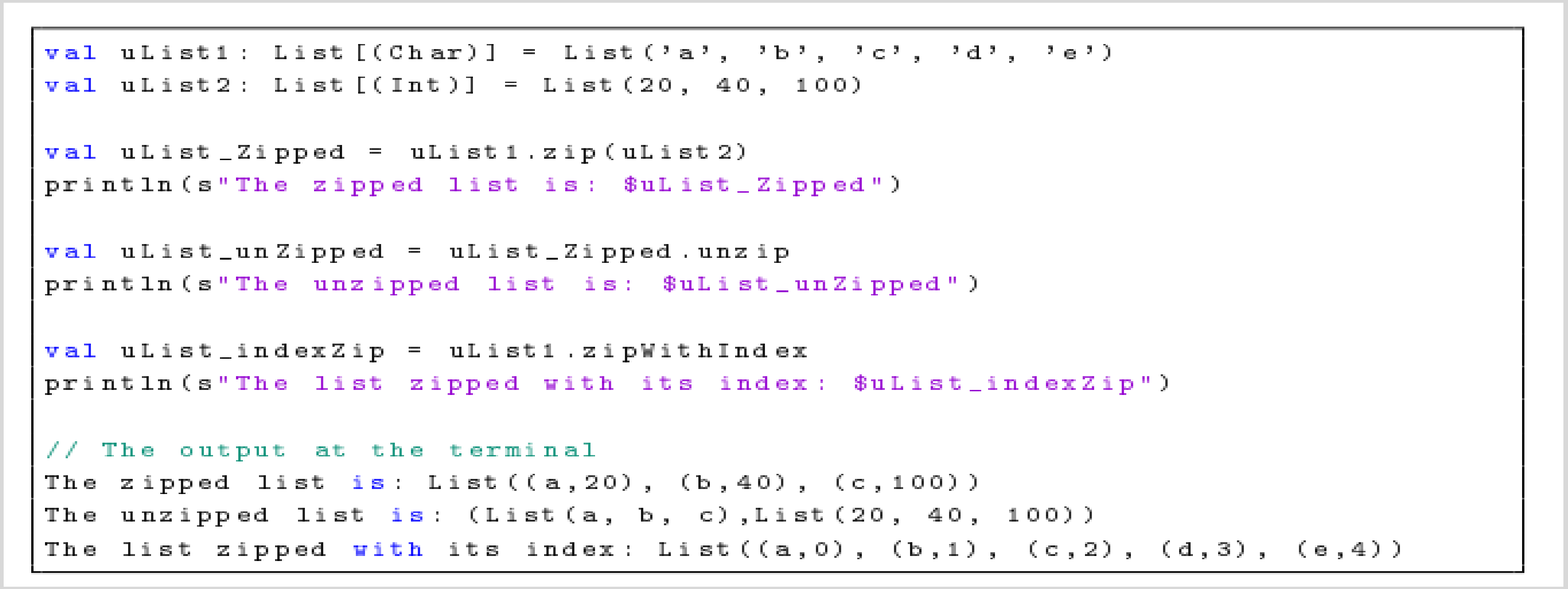
zip and unzip Methods

- The zip method takes two lists as input and creates a list by pairing elements from the two lists with same index. If the lengths of the two lists are different, then unmatched elements are dropped. The zip and unzip methods also allow to operate on multiple lists together. Figure provides the definitions for zip, unzip and zipWithIndex methods.



```
def zip[B](that: IterableOnce[B]): CC[(A, B)] // CC - arbitrary collection
def unzip[A1, A2](implicit asPair: (A) -> (A1, A2)): (CC[A1], CC[A2])
def zipWithIndex: CC[(A, Int)]
```

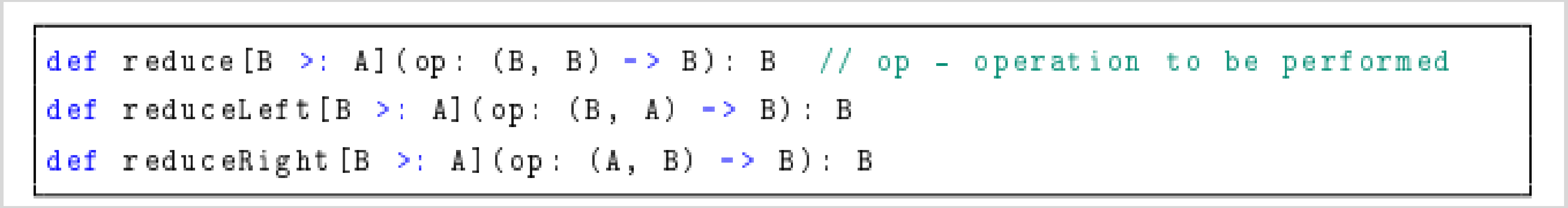
- The zip and unzip methods are applicable to both immutable as well as mutable collection types. The keyword CC in figure represents an arbitrary collection type. Figure 2 illustrates how to apply the zip/unzip methods on two lists with different data types. First, both lists are zipped together after which, the zipped list is unzipped. At the end zipping of list with its index is also given, where each element in the list is paired with its respective index.



```
val uList1: List[(Char)] = List('a', 'b', 'c', 'd', 'e')
val uList2: List[(Int)] = List(20, 40, 100)

val uList_Zipped = uList1.zip(uList2)
println(s"The zipped list is: $uList_Zipped")

val uList_unZipped = uList_Zipped.unzip
println(s"The unzipped list is: $uList_unZipped")

val uList_indexZip = uList1.zipWithIndex
println(s"The list zipped with its index: $uList_indexZip")

// The output at the terminal
The zipped list is: List((a,20), (b,40), (c,100))
The unzipped list is: (List(a, b, c),List(20, 40, 100))
The list zipped with its index: List((a,0), (b,1), (c,2), (d,3), (e,4))
```

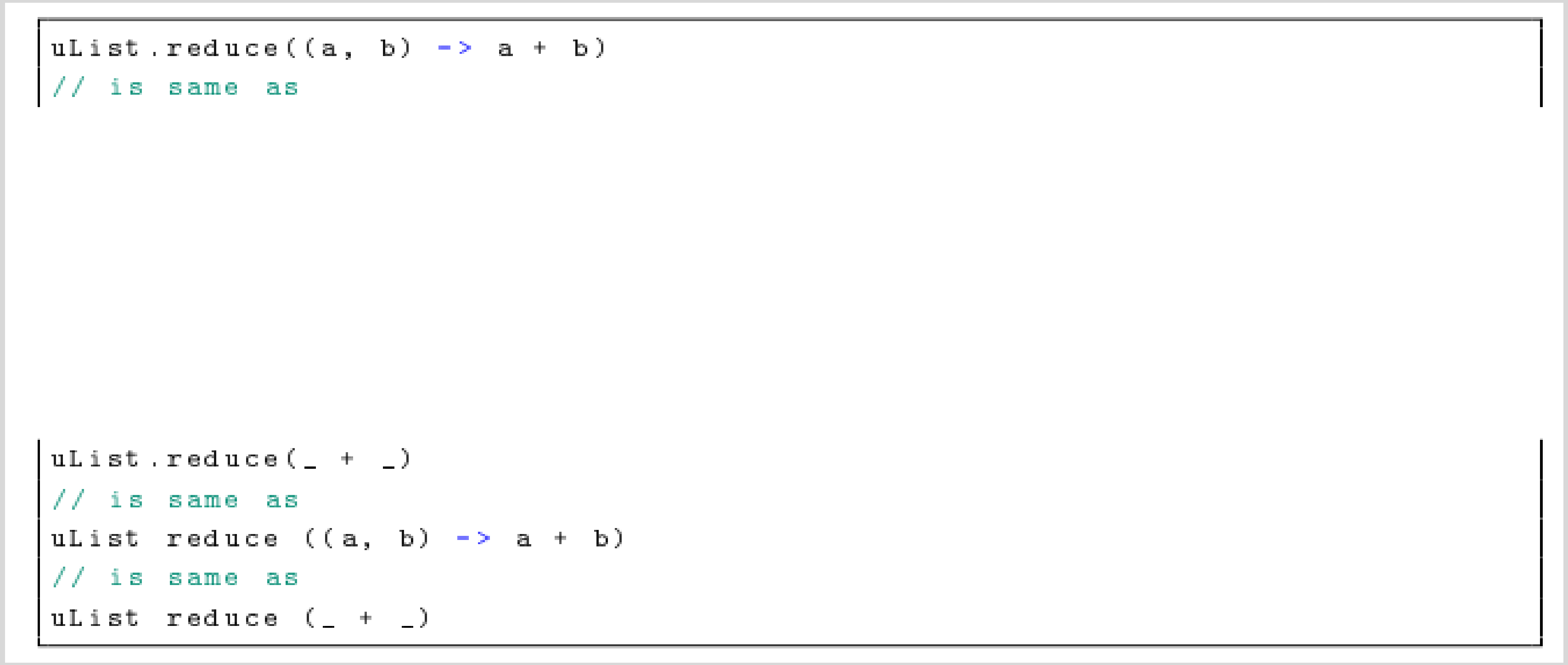
The reduce Method

- The reduce is a higher order method available with many Scala collections. This method traverses all the elements in a collection and combines them in some way (specified in the argument) to produce the result. The reduce method uses binary operations to combine the elements and can not be applied to an empty collection. It is worth mentioning that the order of applying binary operations does matter. Figure provides the definitions for reduce, reduceLeft and reduceRight methods.



```
def reduce[B >: A](op: (B, B) => B): B  // op - operation to be performed
def reduceLeft[B >: A](op: (B, A) => B): B
def reduceRight[B >: A](op: (A, B) => B): B
```

◦ The syntactic sugar syntax for reduce method is illustrated in figure

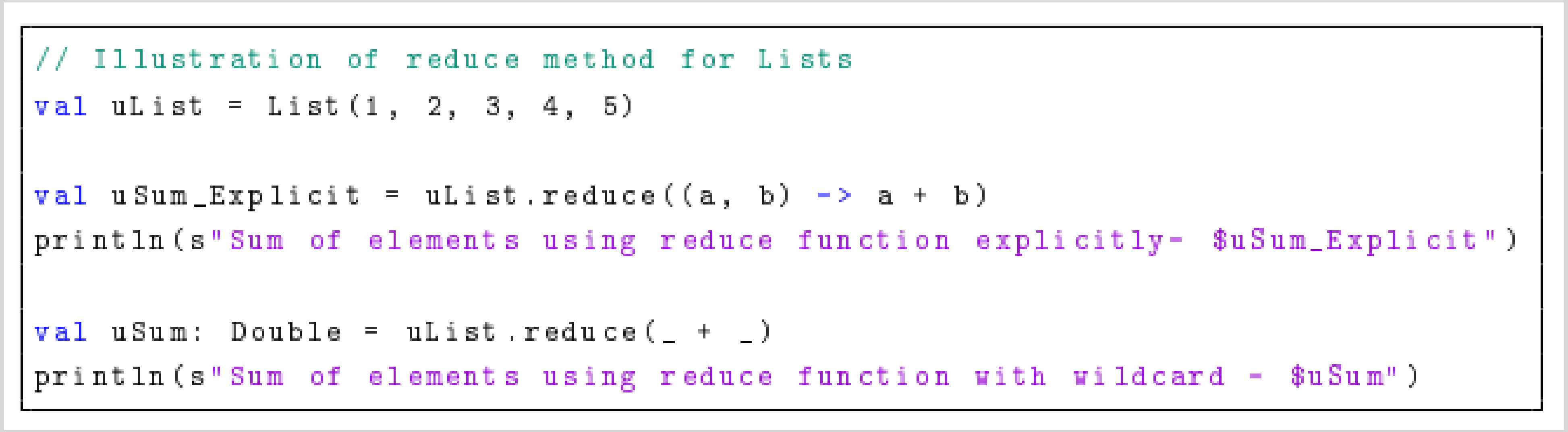


```
uList.reduce((a, b) -> a + b)
// is same as
```

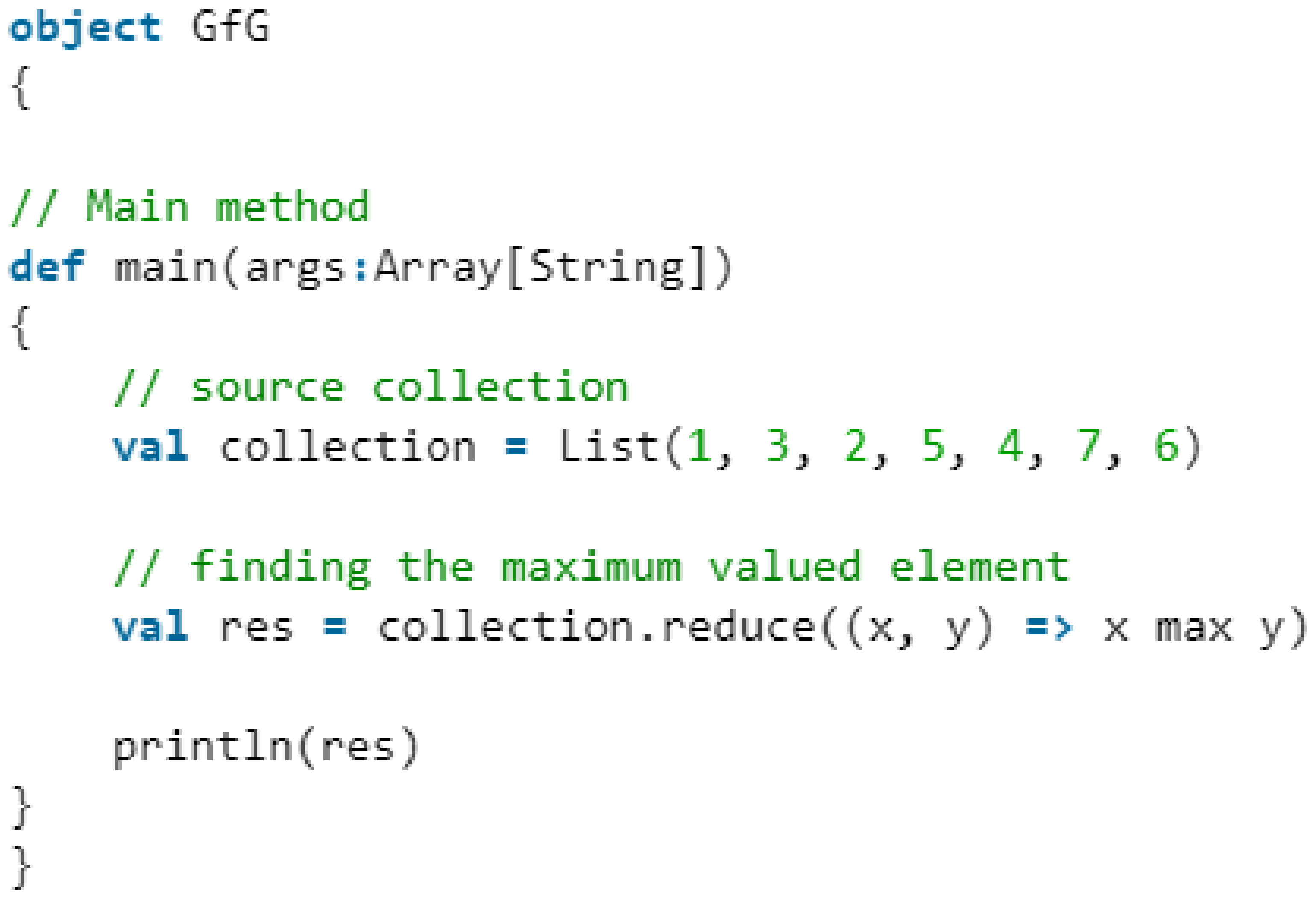
```
uList.reduce(_ + _)
// is same as
uList reduce ((a, b) -> a + b)
// is same as
uList reduce (_ + _)
```

reduce Method: Examples

- Let us illustrate the working of reduce method using some examples. Figure demonstrates the summation of list elements using reduce method. Two instances are given, one with explicit arguments and second using wildcard. Both the cases will produce the same result provided same operation has been applied while invoking the method.

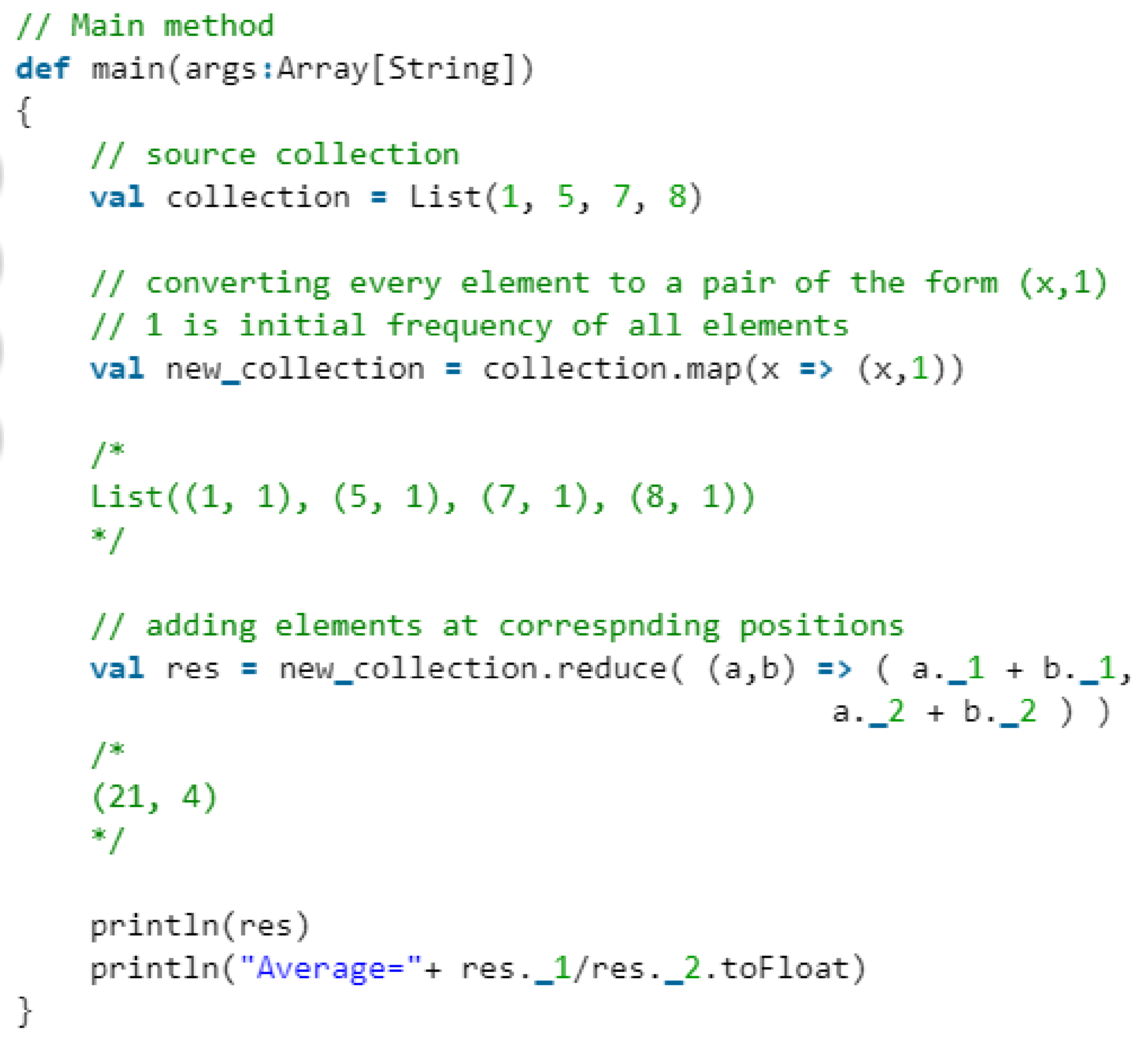


```
// Illustration of reduce method for Lists
val uList = List(1, 2, 3, 4, 5)

val uSum_Explicit = uList.reduce((a, b) -> a + b)
println(s"Sum of elements using reduce function explicitly- $uSum_Explicit")

val uSum: Double = uList.reduce(_ + _)
println(s"Sum of elements using reduce function with wildcard - $uSum")
```



```
object GfG
{

// Main method
def main(args:Array[String])
{
    // source collection
    val collection = List(1, 3, 2, 5, 4, 7, 6)

    // finding the maximum valued element
    val res = collection.reduce((x, y) => x max y)

    println(res)
}
}
```

Example-1
reduce



```
// Main method
def main(args:Array[String])
{
    // source collection
    val collection = List(1, 5, 7, 8)

    // converting every element to a pair of the form (x,1)
    // 1 is initial frequency of all elements
    val new_collection = collection.map(x => (x,1))

    /*
    List((1, 1), (5, 1), (7, 1), (8, 1))
    */

    // adding elements at correspnding positions
    val res = new_collection.reduce( (a,b) => ( a._1 + b._1,
                                              a._2 + b._2 ) )
    /*
    (21, 4)
    */

    println(res)
    println("Average="+ res._1/res._2.toFloat)
}
```

The second example shown in figure instantiates a list and then creates a modified one that pairs each element of the first list with 1. Afterwards reduce method is applied on the compound list and it is explicitly mentioned in the argument that what operation is performed on the first elements of each pair and correspondingly on the second elements. This reduces the paired list to a single pair whose elements are operated to evaluate the average.

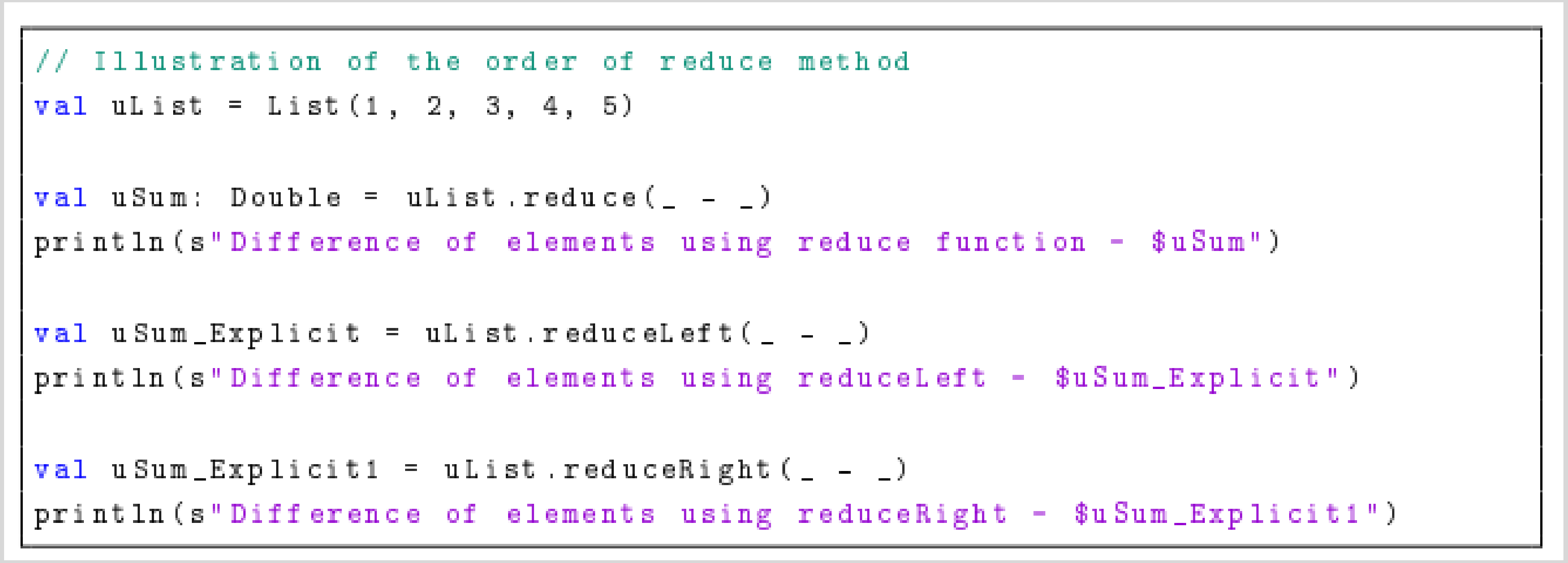
OUTPUT

```
(21, 4)
Average= 5.25
```

- In the above program, all elements of the collection are transformed into tuples with two elements. First element of the tuple is the number itself and the second element is the counter. Initially all counters are set to 1. The output itself is a tuple with two elements: first value is the sum and the second value is the number of elements.
- Note: Type of output given by reduce() method is same as the type of elements of the collection.

Order of reduce Method

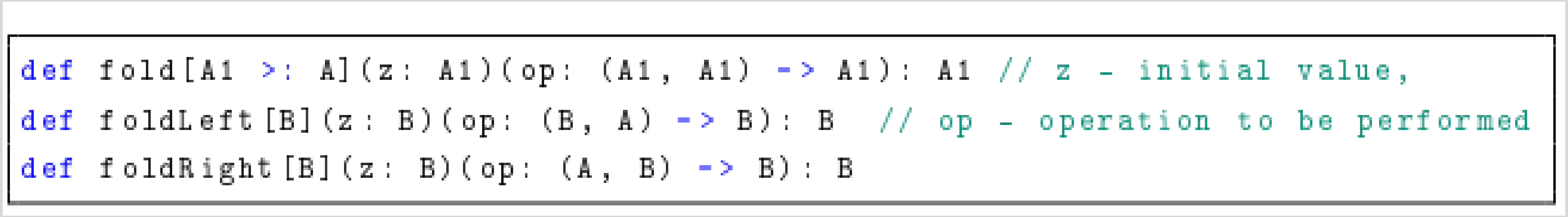
- As mentioned before, the order in which the binary operations are applied by the reduce method, matters. There are two methods, that expand on the functionality of the reduce method, namely, reduceLeft and reduceRight. As their names suggest, the former would yield (a # b) and the latter would be (b # a) where '#' is an arbitrary binary operator. Refer to figure for an application where the order makes a difference in the result.



```
// Illustration of the order of reduce method
val uList = List(1, 2, 3, 4, 5)

val uSum: Double = uList.reduce(_ - _)
println(s"Difference of elements using reduce function - $uSum")

val uSum_Explicit = uList.reduceLeft(_ - _)
println(s"Difference of elements using reduceLeft - $uSum_Explicit")

val uSum_Explicit1 = uList.reduceRight(_ - _)
println(s"Difference of elements using reduceRight - $uSum_Explicit1")
```

fold, foldLeft and foldRight Methods

- The fold method is used to collapse the elements of a collection using associative binary operator (function). The fold method takes two arguments; the initial value and the function. The function itself takes two arguments; the function result from the previous iteration and the current item from the list. For the first iteration, the function result takes the initial value. The figure provides definitions for fold, foldLeft and foldRight methods.



```
def fold[A1 >: A](z: A1)(op: (A1, A1) -> A1): A1 // z - initial value,
def foldLeft[B](z: B)(op: (B, A) -> B): B  // op - operation to be performed
def foldRight[B](z: B)(op: (A, B) -> B): B
```

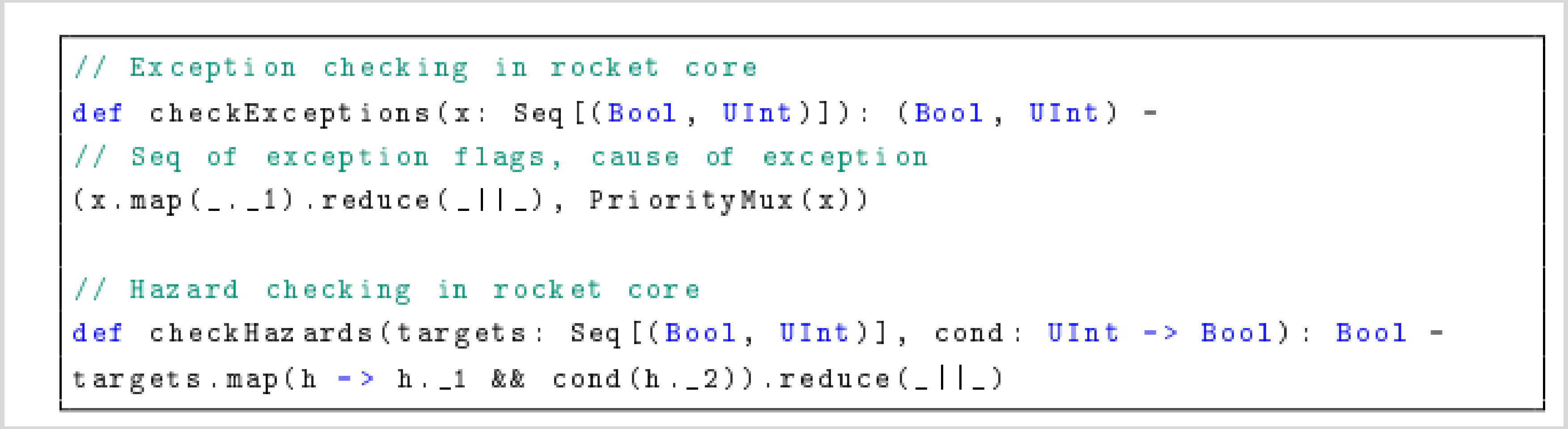
```scala
// source collection
val uList = List(1, 5, 7, 8)

// converting every element to a pair of the form (x,1)
val uList_Modified = uList.map(x => (x, 1))

// adding elements at correspnding positions
val result = uList_Modified.fold(0,0)((a, b) => (a._1 + b._1, a._2 + b._2))
val average = (result._1).toFloat / (result._2).toFloat
```

```scala
println("(sum, no_of_elements) - " + result)
println("Average - " + average)

// The result at the terminal
(sum, no_of_elements) = (21,4)
Average = 5.25
```

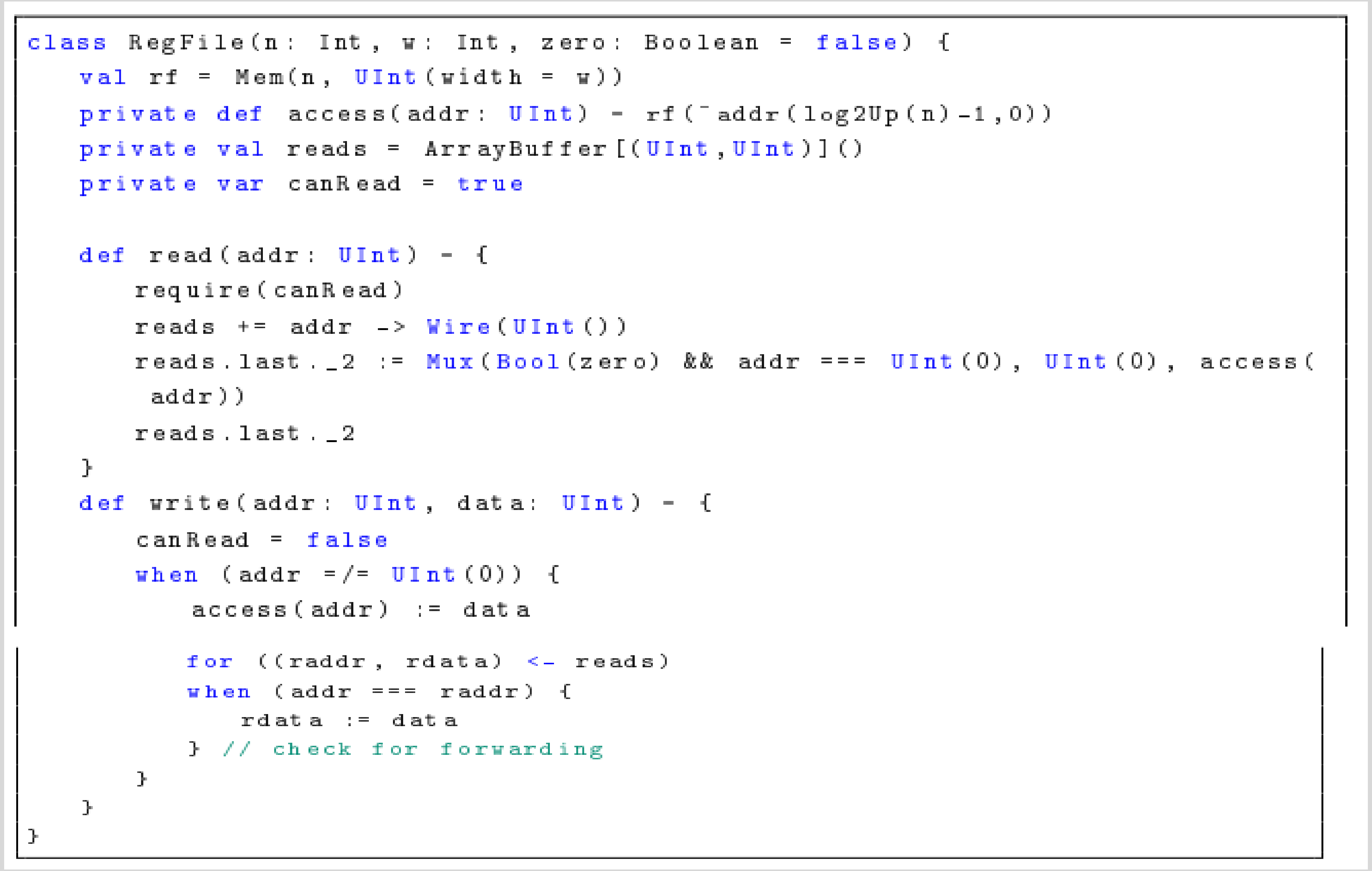
We repeat the example in figure, by only replacing the reduce method with fold method.

Illustrations from Rocket Chip

- Different methods discussed above are further illustrated using selected examples from Rocket1 chip generator. The first example defines an exception (checkExceptions) and hazard (checkHazards) checking methods as outlined in figure. The map and reduce methods are applied, to a Seq of Tuple2, to find out if there is an exception/hazard or not? Also note the return types of these user defined methods.



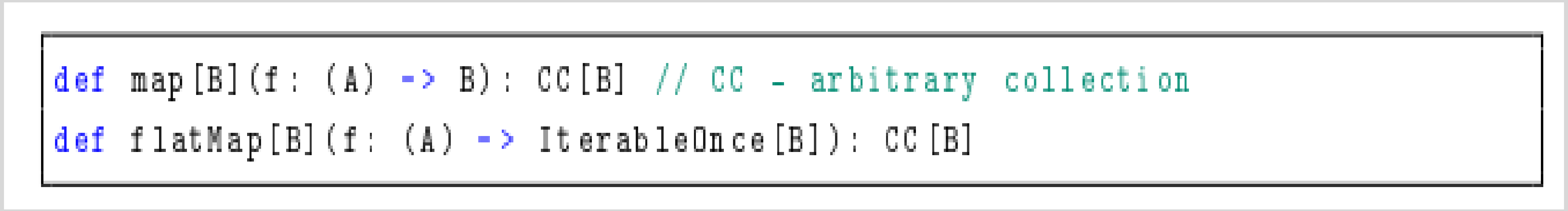
```
// Exception checking in rocket core
def checkExceptions(x: Seq[(Bool, UInt)]): (Bool, UInt) -
// Seq of exception flags, cause of exception
(x.map(_._1).reduce(_||_), PriorityMux(x))

// Hazard checking in rocket core
def checkHazards(targets: Seq[(Bool, UInt)], cond: UInt -> Bool): Bool -
targets.map(h -> h._1 && cond(h._2)).reduce(_||_)
```



```
class RegFile(n: Int, w: Int, zero: Boolean = false) {
  val rf = Mem(n, UInt(width = w))
  private def access(addr: UInt) - rf(~addr(log2Up(n)-1,0))
  private val reads = ArrayBuffer[(UInt,UInt)]()
  private var canRead = true

  def read(addr: UInt) - {
    require(canRead)
    reads += addr -> Wire(UInt())
    reads.last._2 := Mux(Bool(zero) && addr === UInt(0), UInt(0), access(
     addr))
    reads.last._2
  }
  def write(addr: UInt, data: UInt) - {
    canRead = false
    when (addr =/= UInt(0)) {
      access(addr) := data
      for ((raddr, rdata) <- reads)
      when (addr === raddr) {
        rdata := data
      } // check for forwarding
    }
  }
}
```

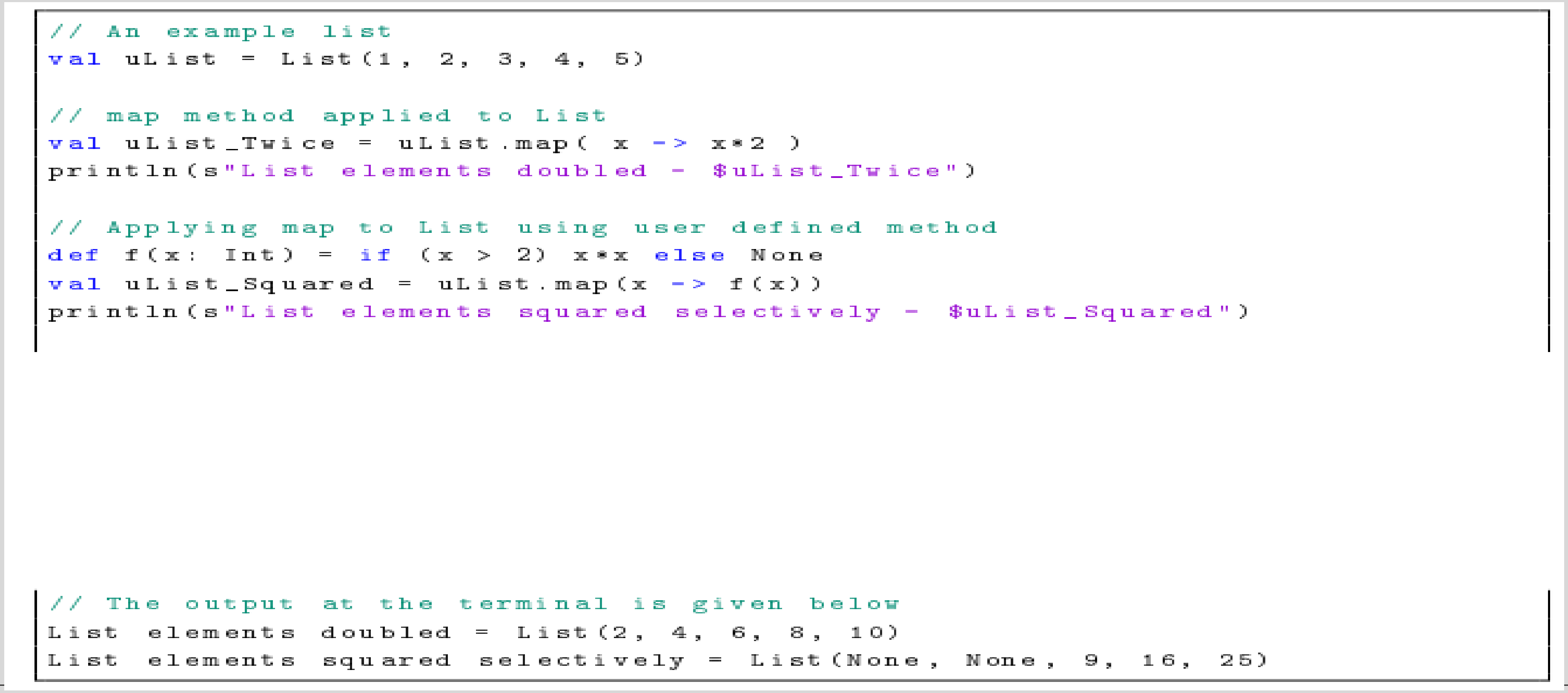
Map, map and flatMap

◦ The Map is, by default, an immutable collection in Scala. As discussed previously, it is a collection of key-value pairs, which is similar to a dictionary. Key-value pairs, in Map, can have an arbitrary data type. However, once the data type has been used for a key and the associated value, then its consistency must be maintained throughout. On the other hand map and flatMap are methods that can be applied to different collections including Map collection. The map is a functional mapping applied to each element of the collection. Similarly, the flatMap is also a functional mapping applied to each element but it flattens the results as will be illustrated later



```
def map[B](f: (A) -> B): CC[B] // CC - arbitrary collection
def flatMap[B](f: (A) -> IterableOnce[B]): CC[B]
```

- the use of map method applied to a list. The function used for mapping can be mentioned either explicitly or can have the form of a user defined function.



```
// An example list
val uList = List(1, 2, 3, 4, 5)

// map method applied to List
val uList_Twice = uList.map( x => x*2 )
println(s"List elements doubled - $uList_Twice")

// Applying map to List using user defined method
def f(x: Int) = if (x > 2) x*x else None
val uList_Squared = uList.map(x => f(x))
println(s"List elements squared selectively - $uList_Squared")
```

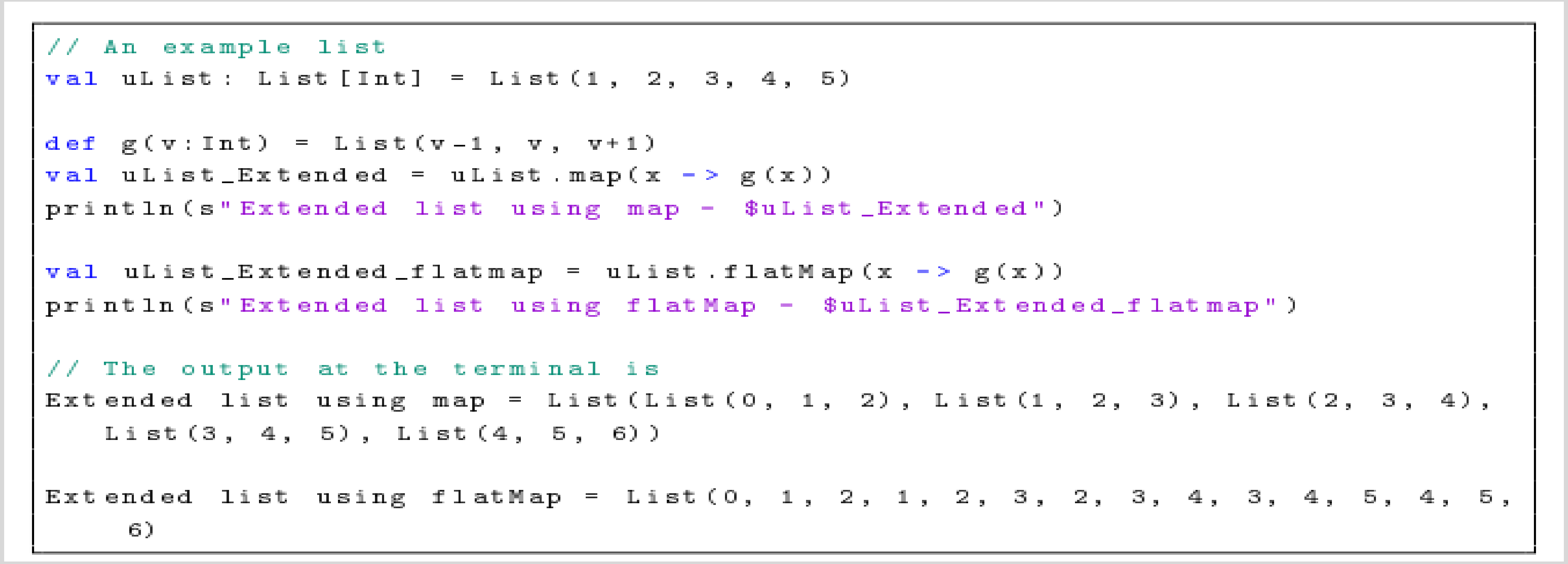
```
// The output at the terminal is given below
List elements doubled = List(2, 4, 6, 8, 10)
List elements squared selectively = List(None, None, 9, 16, 25)
```

Important points about map() method:

- map() is a higher order function.
- Every collection object has the map() method.
- map() takes some function as a parameter.
- map() applies the function to every element of the source collection.
- map() returns a new collection of the same type as the source collection.

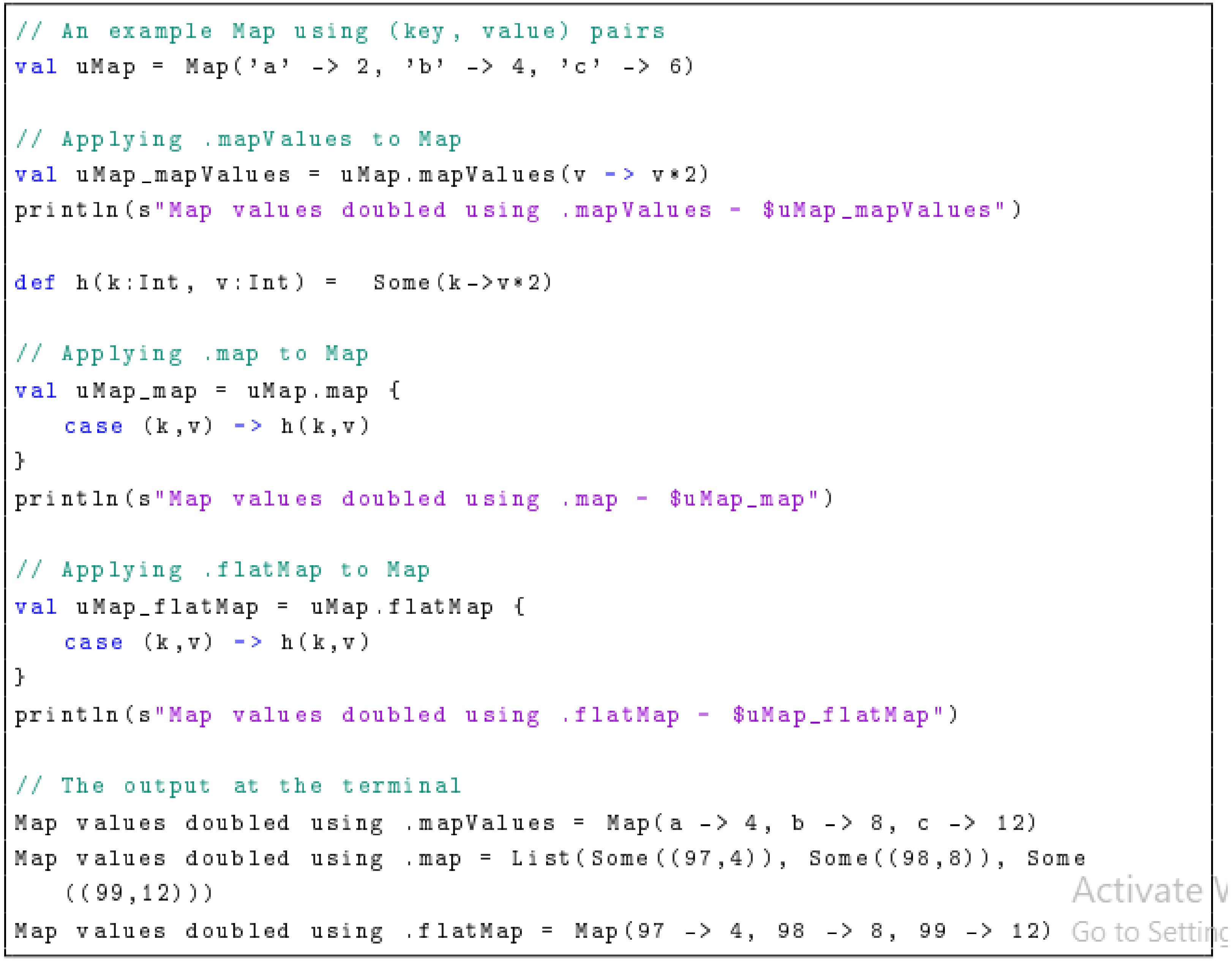
```
// square of an integer
   def square(a:Int):Int
   =
   {
      a*a
   }
// Main method
   def main(args:Array[String])
   {
      // source collection
      val collection = List(1, 3, 2, 5, 4, 7, 6)

      // transformed collection
      val new_collection=collection.map(square)
      println(new_collection)
   }}
```

- Next we compare and contrast the working of map and flatMap methods. For that purpose, we apply the map and flatMap to the same list by employing the user defined function g(v:Int) as discussed in figure. The flatMap not only flattens the list but also eliminates any nonexistent (empty) entries from the resulting list as can be seen from the



```
// An example list
val uList: List[Int] = List(1, 2, 3, 4, 5)

def g(v:Int) = List(v-1, v, v+1)
val uList_Extended = uList.map(x -> g(x))
println(s"Extended list using map - $uList_Extended")

val uList_Extended_flatmap = uList.flatMap(x -> g(x))
println(s"Extended list using flatMap - $uList_Extended_flatmap")

// The output at the terminal is
Extended list using map = List(List(0, 1, 2), List(1, 2, 3), List(2, 3, 4),
   List(3, 4, 5), List(4, 5, 6))

Extended list using flatMap = List(0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5,
    6)
```

```
// An example list
val uList: List[Int] = List(1, 2, 3, 4, 5)

// Applying map and flatMap to List with builtin Options class
def f(x: Int) = if (x > 2) Some(x) else None
val uList_selective = uList.map(x -> f(x))
println(s"Selective elements of List with .map - $uList_selective")

val uList_selective_flatMap = uList.flatMap(x -> f(x))
println(s"Selective elements of List with .flatMap -
    $uList_selective_flatMap")

// Output at the terminal
Selective elements of List using .map = List(None, None, Some(3), Some(4),
    Some(5))
Selective elements of List using .flatMap = List(3, 4, 5)
```

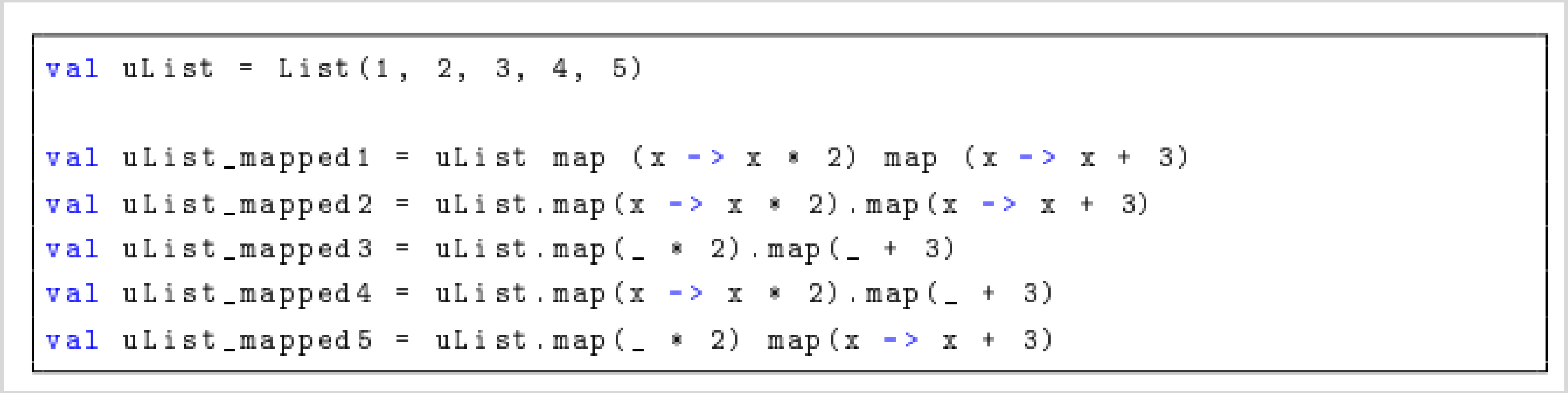


```scala
// An example Map using (key, value) pairs
val uMap = Map('a' -> 2, 'b' -> 4, 'c' -> 6)

// Applying .mapValues to Map
val uMap_mapValues = uMap.mapValues(v -> v*2)
println(s"Map values doubled using .mapValues - $uMap_mapValues")

def h(k:Int, v:Int) =  Some(k->v*2)

// Applying .map to Map
val uMap_map = uMap.map {
    case (k,v) -> h(k,v)
}
println(s"Map values doubled using .map - $uMap_map")

// Applying .flatMap to Map
val uMap_flatMap = uMap.flatMap {
    case (k,v) -> h(k,v)
}
println(s"Map values doubled using .flatMap - $uMap_flatMap")

// The output at the terminal
Map values doubled using .mapValues = Map(a -> 4, b -> 8, c -> 12)
Map values doubled using .map = List(Some((97,4)), Some((98,8)), Some
    ((99,12)))
Map values doubled using .flatMap = Map(97 -> 4, 98 -> 8, 99 -> 12)
```

Finally, we illustrate the use of map and flatMap with the Map type collections

Map, map and flatMap

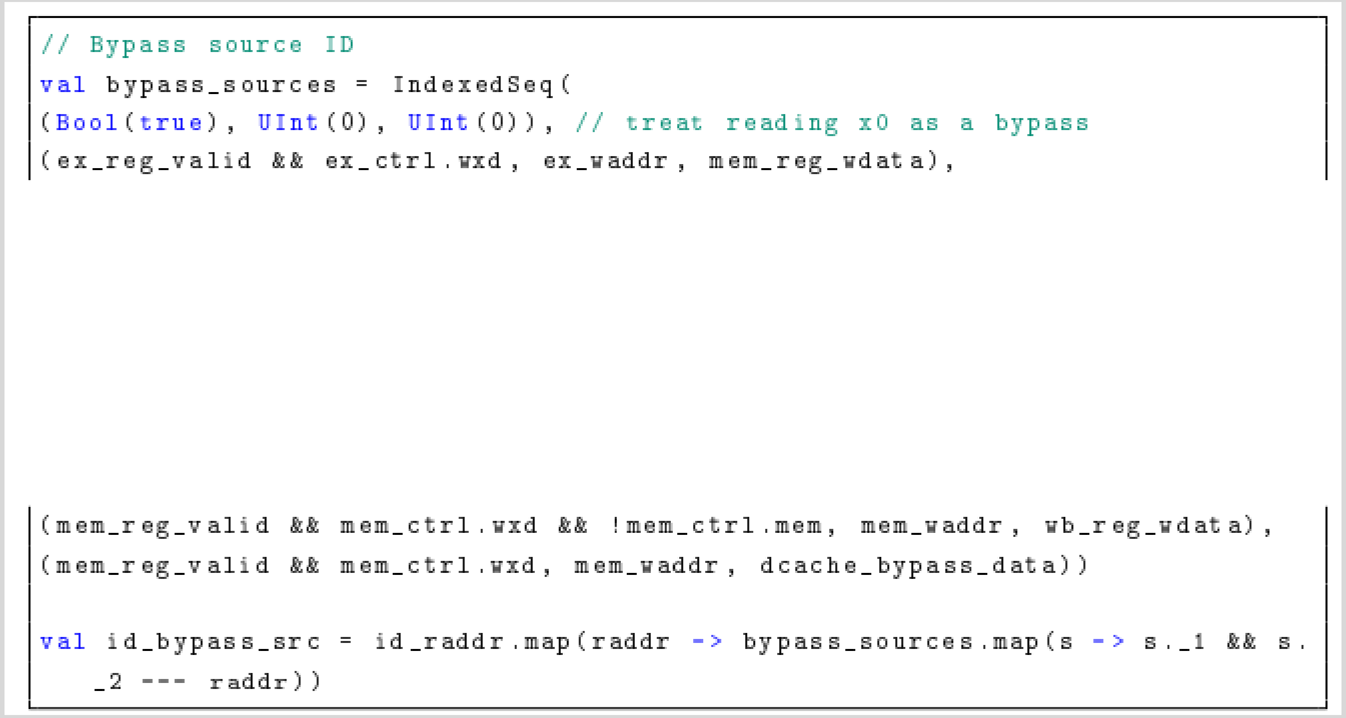
- illustrates different syntax variants using syntactic sugar, for map method, while generating the same result.



```
val uList = List(1, 2, 3, 4, 5)

val uList_mapped1 = uList map (x -> x * 2) map (x -> x + 3)
val uList_mapped2 = uList.map(x -> x * 2).map(x -> x + 3)
val uList_mapped3 = uList.map(_ * 2).map(_ + 3)
val uList_mapped4 = uList.map(x -> x * 2).map(_ + 3)
val uList_mapped5 = uList.map(_ * 2) map(x -> x + 3)
```

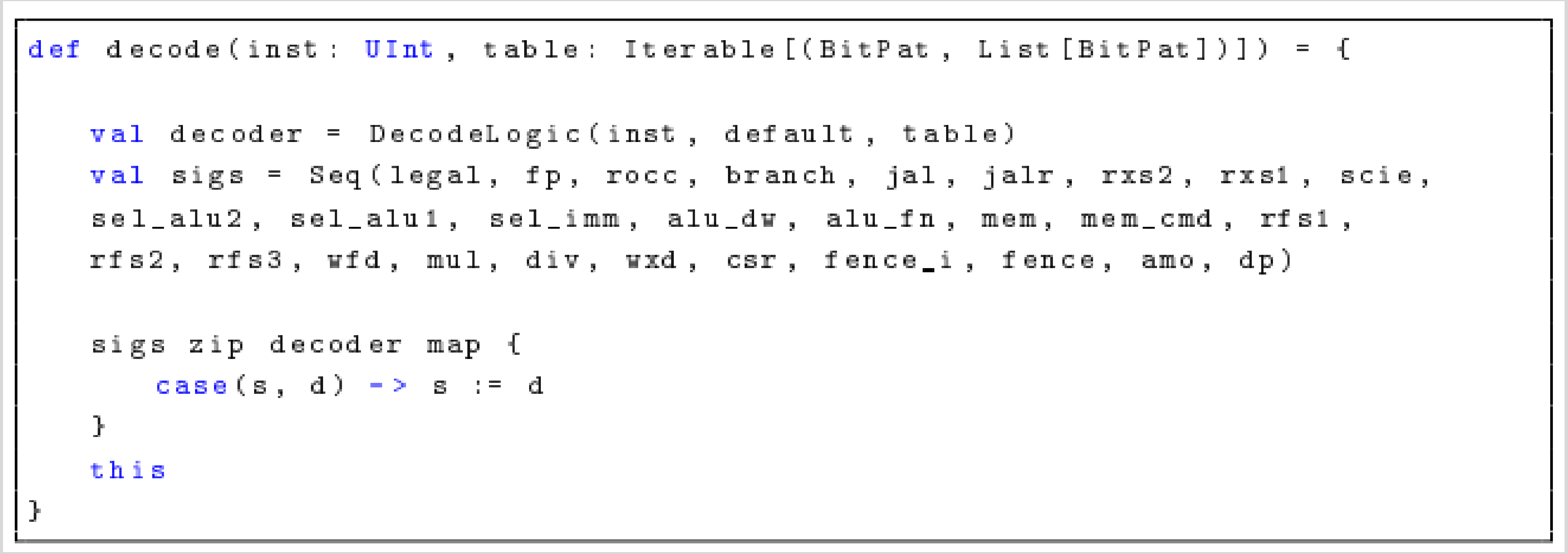
Illustrations from Rocket Chip

- The first example determines the bypass source using map method and is given in figure. An indexed sequence is used to form a collection of bypass sources.



```
// Bypass source ID
val bypass_sources = IndexedSeq(
(Bool(true), UInt(0), UInt(0)), // treat reading x0 as a bypass
(ex_reg_valid && ex_ctrl.wxd, ex_waddr, mem_reg_wdata),








(mem_reg_valid && mem_ctrl.wxd && !mem_ctrl.mem, mem_waddr, wb_reg_wdata),
(mem_reg_valid && mem_ctrl.wxd, mem_waddr, dcache_bypass_data))

val id_bypass_src = id_raddr.map(raddr -> bypass_sources.map(s -> s._1 && s.
   _2 --- raddr))
```

- The generation of control signals, using decode method, is illustrated in figure. The use syntactic sugar syntax for zip and map methods can be noticed, when assigning the generated control signals to the IO signals.



```
def decode(inst: UInt, table: Iterable[(BitPat, List[BitPat])]) = {

    val decoder = DecodeLogic(inst, default, table)
    val sigs = Seq(legal, fp, rocc, branch, jal, jalr, rxs2, rxs1, scie,
    sel_alu2, sel_alu1, sel_imm, alu_dw, alu_fn, mem, mem_cmd, rfs1,
    rfs2, rfs3, wfd, mul, div, wxd, csr, fence_i, fence, amo, dp)

    sigs zip decoder map {
        case(s, d) => s := d
    }
    this
}
```