

2.0 Let's write some code

Copyright © Robin Hillyard



Northeastern
University

Hello World!

“Hello World!”

- That’s a perfectly fine Scala expression. But it’s not very useful as is because, if we do nothing else, we will never actually see anything.
- But that’s because we never really specified how we wanted to “see” the string.
- If we were writing a microservice, we would probably want to pack that string into an HTTP response object (or similar).
- But let’s just follow the convention and write it to the console:

```
println (“Hello World!”)
```

Organizing our code for the compiler

- That last bit of code still isn't really useful if we want to compile it because the compiler needs a little bit of “boilerplate”:

```
object HelloWorld {  
  println ("Hello World!")  
}
```

- This tells the compiler that we are defining an “object” and, within it, we have created an expression which prints the message. However, this code will actually do nothing at all (!) because it's not a *program*.
- However, if you were running the “REPL” (read-evaluate-print-loop) you wouldn't need all this object stuff.

Hello World in the REPL

```
Welcome to Scala 2.13.4 (OpenJDK 64-Bit Server VM, Java
11.0.8).
```

```
Type in expressions for evaluation. Or try :help.
```

```
scala> "Hello World!"
```

```
val res0: String = Hello World!
```

```
scala> println ("Hello World!")
```

```
Hello World!
```

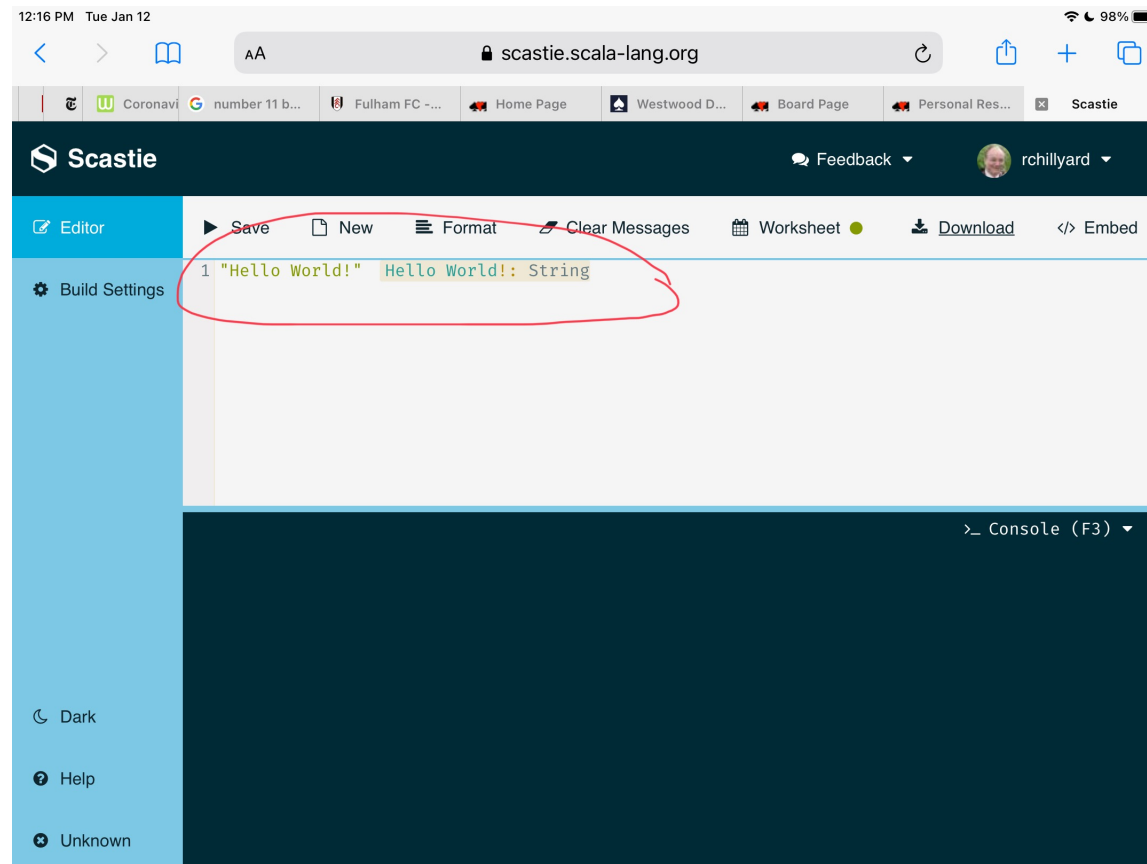
```
scala> println (res0)
```

```
Hello World!
```

```
scala>
```

Scastie

- You can play with Scala on the web at <https://scastie.scala-lang.org>



Scalafiddle


- You can do the same thing with scalafiddle, even better: it allows you to easily include functional libraries, such as Cats, ScalaZ, Shapeless, Monocle, etc.
- However, it's stuck on Scala 2.12 (and we should learn 2.13, or 3.0).
- We'll come back to this later.

Back to Scastie...

- Did you notice that we didn't need to say *println* because we saw the evaluated expression in a comment on the right?
- But, normally, we'll want to show things in the console (lower, black, part of the page).
- So, we'll do `println("hello world!")` as before [please follow along with your own Scastie session].

Our program isn't very interesting

- Because it always does the same thing and that thing is pretty simple.
- Let's do something more interesting:

```
val hw = "Hello World!"  
def time = java.time.LocalDateTime.now  
def greet = s"$hw it's $time"  
println(greet)  
println(greet)
```


Variables and methods

- The *val* keyword in front of *hw*, declares it as a “value”— *hw* is really best thought of as an alias for “Hello World!” (More on this later).
- But we call these “variables” in Scala because they serve the same purpose as variables in algebra. They don’t change their value once set up (they are immutable), but they can stand in for anything.
- But what about *time* and *greet*?
- Again, in each case, we have *defined* an alias for an expression. The difference here is that the evaluation of the expression won’t always be the same.
 - *But, in functional programming, we normally try to have methods which always return the same result for a given parameter set. What’s up with this?*

Pure and impure methods

- There are two reasons why a method might not always yield the same result:
 - The method takes parameters (which, of course, can have different values);
 - The method might not be a *pure* method.
- A **pure** method is one which, for a given set of input parameter(s), will always yield the same result AND there must be no side-effects*.
- A less strict kind of method is an **idempotent** method.
 - An idempotent method typically has side-effects, but it can be called any number of times and the effect on the rest of the world will be the same as calling it only once.

* See next slide

Effects

- Functional programs are all about *expressions*.
 - But, if we don't actually use that expression anywhere, it will not affect the outside world (we are part of the outside world).
 - Therefore, we will never see the result.
 - What use is that?
- So, any real-life computer program must cause some *effect* to happen in the outside world.
 - How can we do this functionally?
- In a pure functional language, such as Haskell, then all I/O (input/output) is achieved through *effects*.
- But, Scala is a lot less strict about effects.

Side-effects

- A method which obviously is designed to cause an effect, such as *println*, is not—in and of itself—a side-effect.
 - That's because *println* yields *Unit* (the empty result type) so a programmer just knows that, if the method is to achieve anything at all, it must be through an effect—and the very name of it makes that obvious.
- But, a method which logs some string *and* yields a result, is causing a *side-effect*.
 - Side-effects are bad because we can't easily reason about them. And, if we can't reason about them, we can't prove a program that involves them.
- Just imagine testing the system clock.
 - Invoking, e.g. *System.currentTimeMillis()*, relies on an effect (the ticking of the system clock).
 - How could you ever test it? And you definitely can't prove such a program.

A Side-Effect (often just called Effect) is everything else. I.e. everything that is not reading the arguments and returning a result is a Side-Effect.

Back to Hello World

- We have embedded the current time into our greeting.
- The time method has no parameters but it's *not pure*: it relies on an independent source of information: the system clock.
- That's why we should define it using “def” instead of “val.” A **val** is evaluated only once, while a **def** is evaluated every time it is invoked (which makes more sense when there are varying parameters).
- It may come as a bit of a surprise that we can defined a “def” (i.e. a method) without any parameters at all.
 - By convention*, if the method relies on a side-effect (i.e. not pure), then we define it (and call it) with empty parentheses—example: **close()**.
 - if the method is pure, we define (and call) it without parentheses.

* The compiler doesn't get too upset if we get this wrong.



括号()

Other details

- We didn't have to specify the type of the *val* and **defs**: that's because Scala has type inference.

```
val hw = "Hello World!"  
def time() = java.time.LocalDateTime.now()  
def greet() = s"$hw it's ${time()}"  
println(greet())  
println(greet())
```

- *Time()* relies on a Java class and is not pure. There's no problem with that.
- *greet* relies on *string interpolation* (the *s"..."* construct).
- Each time we invoke *greet*, the result is slightly different.
- BTW, I've tried to emphasize the keyword/tokens by putting them in **bold**.

Back to using the compiler

- If we want to compile this program and be able to run it in order to get the time, we are going to have to put the boiler plate back...

```
object HelloWorld extends App {  
  val hw = "Hello World!"  
  def time = java.time.LocalDateTime.now  
  def greet = s"$hw it's $time"  
  println(greet)  
  println(greet)  
}
```

- Notice the “extends App” in the definition of HelloWorld. Without that, still nothing would happen because there is no “main” program.
- *App* is a class which causes the (non-declaration) code to be run when you ask to run the program.

What about control structures?

- Other than declarations, there really aren't any statements in Scala code. Everything is based on expressions.
- So, it probably won't come as a big surprise that control structures are all basically functions which yield a value and which are invoked via keywords. For example:

```
val msg = if (time.getDayOfWeek.getValue==0) "happy rest day" else "keep working"
```

- Or, we could write:

```
val message = time.getDayOfWeek match {  
  case java.time.DayOfWeek.MONDAY => "Monday's child is fair of face"  
  case java.time.DayOfWeek.TUESDAY => "Tuesday's child is full of grace"  
  case java.time.DayOfWeek.WEDNESDAY => "Wednesday's child is full of woe"  
  case java.time.DayOfWeek.THURSDAY => "Thursday's child has far to go"  
  case _ => "I didn't have enough space to do these"  
}
```

- Note how all of these are expressions which yield a result.

Imports

- BTW, there's another kind of aliasing that we haven't come across yet. It's invoked when we use *import*.

```
import java.time.LocalDateTime
import java.time.DayOfWeek._
object HelloWorld extends App {
  val hw = "Hello World!"
  def time = LocalDateTime.now
  def greet = s"$hw it's $time"
  println(greet)
  println(greet)
  val msg = if (time.getDayOfWeek.getValue==0) "happy rest day" else "keep working"
  val message = time.getDayOfWeek match {
    case MONDAY => "Monday's child is fair of face"
    case TUESDAY => "Tuesday's child is full of grace"
    case WEDNESDAY => "Wednesday's child is full of woe"
    case THURSDAY => "Thursday's child has far to go"
    case _ => "I didn't have enough space to do these"
  }
}
```

What if we want to get the clock time several repeatedly?

- There are a couple of different ways, depending on whether we want to invoke something with effects (like printing on the console) or we want to get a list of results:

```
(0 until 3) foreach ( _ => println(greet))
```

```
for ( _ <- 0 until 3) println(greet)
```

```
val greetings = for ( _ <- 0 until 3) yield greet
```

lambda



- The first two are pretty much equivalent. Notice the "yield" in the third one. Each results in three greetings (but the last doesn't print anything).
- BTW, you might notice the use of "_" in each of these expressions. This basically represents an unidentified variable, that's to say a variable whose name we don't need because we never are going to refer to it.

Anything else?

- I'm told there's a **do while** construct but you should never use it!
- And there's an **if** clause which, like **for** without **yield**, conditionally invokes a side-effect. But you really shouldn't use it.
- Note that the normal **if** expression is like "?" in Java. So, you must always provide an **else** expression (if you see a compiler message about *Any* type or perhaps *Serialized* or *Product* type, check your **ifs**).
- Are you wondering if it's possible to declare mutable variables?
 - Good question. We'll discuss in much more detail later.
 - There are such things—but you should never use them!