Scala Puzzlers

Excerpt

artima

ARTIMA PRESS
WALNUT CREEK, CALIFORNIA

Buy the Book · Discuss

Puzzle 2

Hi There!

Scala places a strong emphasis on writing simple, concise code. Its syntax for anonymous functions, `arg => expr`, makes it easy to construct function literals with minimal boilerplate, even when the functions consist of multiple statements.

For functions with self-explanatory arguments, you can do better and use placeholder syntax. This trims away the argument declaration. For example:

```
List(1, 2).map { i => i + 1 }
```

becomes:

```
List(1, 2).map { _ + 1 }
```

The two are equivalent:

```
scala> List(1, 2).map { i => i + 1 }
res1: List[Int] = List(2, 3)

scala> List(1, 2).map { _ + 1 }
res0: List[Int] = List(2, 3)
```

What if you added a debugging statement to the above example to help you understand when the function is applied? What is the result of executing the following code in the REPL?

```
List(1, 2).map { i => println("Hi"); i + 1 }
List(1, 2).map { println("Hi"); _ + 1 }
```

Possibilities

1. Prints:

   ```
   Hi
   List[Int] = List(2, 3)
   Hi
   List[Int] = List(2, 3)
   ```

2. Prints:

   ```
   Hi
   Hi
   List[Int] = List(2, 3)
   Hi
   Hi
   List[Int] = List(2, 3)
   ```

3. Prints:

   ```
   Hi
   Hi
   List[Int] = List(2, 3)
   Hi
   List[Int] = List(2, 3)
   ```

4. The first statement prints:

   ```
   Hi
   Hi
   List[Int] = List(2, 3)
   ```

   and the second fails to compile.

Explanation

You need not be concerned with compiler errors, because the code compiles without problems; yet, it does not behave the way you might expect. The correct answer is number 3:

```
scala> List(1, 2).map { i => println("Hi"); i + 1 }
Hi
Hi
res23: List[Int] = List(2, 3)

scala> List(1, 2).map { println("Hi"); _ + 1 }
Hi
res25: List[Int] = List(2, 3)
```

What is going on here? If the function with the explicit argument prints `Hi` twice, as it is invoked for each element in the list, why doesn't our function with placeholder syntax do the same?

Since anonymous functions are often passed as arguments, it's common to see them surrounded by `{ ... }` in code. It's easy to think that these curly braces represent an anonymous function, but instead they delimit a *block expression*: one or multiple statements, with the last determining the result of the block.

The way the two code blocks are parsed determines the difference in behavior. The first statement, `{ i => println("Hi"); i + 1 }`, is identified as *one* function literal expression of the form `arg => expr`, with `expr` here being the block, `println("Hi"); i + 1`. Since the `println` statement is part of the function body, it is executed each time the function is invoked.

```
scala> val printAndAddOne =
    (i: Int) => { println("Hi"); i + 1 }
printAndAddOne: Int => Int = <function1>

scala> List(1, 2).map(printAndAddOne)
Hi
Hi
res29: List[Int] = List(2, 3)
```

In the second statement, however, the code block is identified as *two* expressions: `println("Hi")` and `_ + 1`. The block is executed, and the last expression (which is conveniently of the required function type, specifically, `Int => Int`) is passed to `map`. The `println` statement is not part of the function body. It is invoked when the argument to `map` is *evaluated*, not as part of the *execution* of `map`.

```
scala> val printAndReturnAFunc =
    { println("Hi"); (_: Int) + 1 }
Hi
printAndReturnAFunc: Int => Int = <function1>

scala> List(1, 2).map(printAndReturnAFunc)
res30: List[Int] = List(2, 3)
```

Discussion

The key lesson here is that the scope of an anonymous function defined using placeholder syntax stretches *only* to the expression containing the underscore (_). This differs from a "regular" anonymous function, whose body contains everything from the rocket symbol (=>) to the end of the code block. Here's an example:

```
scala> val regularFunc =
    { a: Any => println("foo"); println(a); "baz" }
regularFunc: Any => String = <function1>

scala> regularFunc("hello")
foo
hello
res42: String = baz
```

It's as though a function with placeholder syntax is "confined" to its own code block. For example, the following two functions are equivalent:

```
scala> val anonymousFunc =
    { println("foo"); println(_: Any); "baz" }
foo
anonymousFunc: String = baz
```

```
scala> val confinedFunc =
    { println("foo"); { a: Any => println(a) }; "baz" }
foo
confinedFunc: String = baz
```

Scala encourages concise code, but there is such a thing as *too* much conciseness. When using placeholder syntax, be aware of the scope of the function that is created.