Appendix B

Scala Syntax for the Java Programmer

This chapter highlights key differences between Java and Scala syntax. It is intended to get Java programmers quickly up to speed and comfortable writing "Java in Scala"—that is, programs that mostly rely on features available in Java, but written in a Scala syntax.

B.1 A Simple High-Low Game in Java

A program implements a simple guessing game, called *HiLo*. The user picks a secret integer in a given range. The program repeatedly asks yes/no questions until the number is guessed. Interaction is terminal based, and a sample run would resemble this:

```
Playing HiLo between 1 and 10.
Is your number between 1 and 5 ? n
I'm 30% done.
Is your number between 6 and 8 ? maybe
  yes or no? y
I'm 52% done.
Is your number between 6 and 7 ? n
I'm 100% done.
Your number is: 8.
```

In this chapter, the *HiLo* program is implemented separately in Java and in Scala. Both implementations follow the same strategy and offer the same functionalities. The code is stripped of all JavaDoc/ScalaDoc comments and of all assertions. The full implementations are available on the companion website.

B.2 Java Implementation

Let's start with the Java implementation. It begins by importing two small components from Apache Commons:

- the `Validate` class, used to check method arguments and the state of the game;

- the `Range` type, used to represent ranges of integers in the game.

Java
```
import org.apache.commons.lang3.Range;
import org.apache.commons.lang3.Validate;

import java.util.Optional;
import java.util.Scanner;

import static java.lang.Math.log;
```

The class defines two variables `low` and `high` to keep track of the current range of possible numbers, as well as a constant `size`, used to measure progress towards guessing the secret number. A public constructor validates its arguments and initializes the three fields. The constructor is complemented by a static factory method that builds a `HiLo` instance with an implicit low bound of one:

Java
```
public final class HiLo {
  private int low;
  private int high;
  private final int size;

  public HiLo(int min, int max) {
    Validate.isTrue(min <= max, "range cannot be empty");
    Validate.isTrue(min >= 0, "min cannot be negative");
    Validate.isTrue(max < 1_000_000_000, "max must be less than 1_000_000_000");

    low = min;
    high = max;
    size = max - min + 1;
  }

  public static HiLo upto(int max) {
    return new HiLo(1, max);
  }
```

Next, the actual implementation of the game starts. It follows a splitting strategy in which the range of possible numbers is halved based on the user's answer to a *yes/no* question. To implement the splitting, a private method is used to calculate the midpoint of the range:

Java
```
  private int midpoint() {
    return (low + high) / 2;
  }
```

```java
  public Range<Integer> choices() {
    Validate.validState(!solved(), "problem has been solved already");
    return Range.between(low, midpoint());
  }

  public boolean solved() {
    return low == high;
  }

  public void yes() {
    Validate.validState(!solved(), "problem has been solved already");
    high = midpoint();
  }

  public void no() {
    Validate.validState(!solved(), "problem has been solved already");
    low = midpoint() + 1;
  }

  public double progress() {
    return solved() ? 1.0 : 1 - log(high - low + 1) / log(size);
  }

  public int secret() {
    Validate.validState(solved(), "problem not solved yet");
    return low;
  }
```

Note how progress is measured by comparing the logarithm of the size of the current range to the logarithm of the size of the initial range. Logarithms are used to produce a smoother progress function, since the number of steps needed to guess the secret number grows with the logarithm of the size of the initial range.

A helper method `yesOrNo` is implemented to parse the user's answers. It uses a `switch` expression and returns an optional Boolean value, so that a "no result" value can be used for answers that were not in *yes/no* form:

Java

```java
  private static Optional<Boolean> yesOrNo(String line) {
    return switch (line.trim()) {
      case "y", "Y", "yes", "Yes", "YES" -> Optional.of(true);
      case "n", "N", "no", "No", "NO" -> Optional.of(false);
      default -> Optional.empty();
    };
  }
```

A second helper method implements the terminal-based interaction with the user:

```
Java
private static void play(HiLo hilo) {
  Scanner stdin = new Scanner(System.in);

  System.out.printf("Playing HiLo between %d and %d.%n", hilo.low, hilo.high);

  while (!hilo.solved()) {
    var choices = hilo.choices();
    int min = choices.getMinimum();
    int max = choices.getMaximum();
    var question =
        (max > min) ? "Is your number between %d and %d ? ".formatted(min, max)
            : "Is your number %d ? ".formatted(min);
    System.out.print(question);
    var ans = yesOrNo(stdin.nextLine());
    while (ans.isEmpty()) {
      System.out.print("  yes or no? ");
      ans = yesOrNo(stdin.nextLine());
    }
    if (ans.get())
      hilo.yes();
    else
      hilo.no();
    System.out.printf("I'm %.0f%% done.%n", hilo.progress() * 100.0);
  }
  System.out.printf("Your number is: %d.%n", hilo.secret());
}
```

As long as the game is not solved, the user is asked in a loop whether the secret number is in the range returned by method `choices`. An inner loop is used to repeat the question until a satisfactory *yes/no* answer is obtained, at which point the corresponding `yes` or `no` method is called on the game object.

Finally, the main program is implemented. It parses a single command-line argument as a number and initiates a game between 1 and this number:

```
Java
private static void usage() {
  System.out.println("Usage: HiLo <max>");
}

public static void main(String[] args) {
  if (args.length != 1) usage();
  else try {
    int max = Integer.parseInt(args[0]);
    if (max >= 1)
      play(HiLo.upto(max));
      else
        usage();
    } catch (NumberFormatException e) {
      usage();
    }
  }
}
```

B.3 The Same Game in Scala

Presumably, you had no difficulty following the Java implementation of the *HiLo* game. This section now presents a Scala implementation of the exact same game. It follows the same strategy and offers the same features.

As before, the implementation begins by importing additional components. Instead of Apache Commons, the Scalactic library is used to validate method arguments and game states. Scala defines its own `Range` type, so there is no need to import it from an external library:

Scala

```scala
import org.scalactic.Requirements.{ require, requireState }
```

Note the syntax used to import multiple elements from the same package. Also, Scala infers semicolons, which do not need to be added explicitly at the end of lines.

The implementation then proceeds with the definition of a `HiLo` class, like in Java:

Scala

```scala
final class HiLo(min: Int, max: Int):
   require(min <= max, "range cannot be empty")
   require(min >= 0, "min cannot be negative")
   require(max < 1_000_000_000, "max must be less than 1_000_000_000")

   private var low = min
   private var high = max
   private val size = max - min + 1
```

The class definition includes the signature of a *primary* constructor—additional, *auxiliary* constructors can be added. The primary constructor can take zero or more arguments. Note the syntax used to specify types: Scala uses a “`variable: Type`” syntax instead of the “`Type variable`” syntax used in Java. The constructor has no explicit body, but any code that is placed outside methods will be executed at construction time (like non-static initializer blocks in Java). This feature is used here to validate the constructor’s arguments.

The type of the two constructor arguments is `Int`, not `int`. Scala has no notion of primitive types. Conceptually, integers (and Booleans and floating-point numbers) are objects, on which methods can be called:

Scala
```
42.max(10)          // 42
3.14.round          // 3
true.equals(false)  // false
```

The class uses the same two variables (`low` and `high`) and the same constant (`size`) as the Java implementation. Constants are introduced by a keyword `val`, while variables use a keyword `var`. The differences between `val` and `var` hinge on the broader notion of immutability, which is discussed at length in Chapter 3.

A notable difference with Java is that the types of the fields are not specified explicitly and are instead inferred by the compiler. Type inference has always been common in functional languages and has become more prevalent in mainstream languages as well. For instance, Java can now infer some generic type arguments and the types of local variables (but not yet of class fields):

Java
```
List<Date> dates = new ArrayList<>(); // generic <Date> argument of ArrayList inferred
var dates = new ArrayList<Date>();    // local variable type ArrayList<Date> inferred
```

In addition to class fields, type arguments, and local variables, the Scala compiler can also infer the return types of some functions and methods, as demonstrated below. Type inference is discussed in Section 15.5.

Scala has no *static* fields or methods. Instead, it uses a notion of companion object. Every Scala class can have a companion object, and what would be *static* in Java can instead be defined in the companion object of the class. Accordingly, Java's static factory method `upto` is moved to the companion object of the Scala class, along with the other static elements of the Java implementation (e.g., `play` and `main`), as described below.

Class `HiLo` then implements the same core methods as in the Java variant:

Scala
```
   private def midpoint = (low + high) / 2

   def choices: Range =
      requireState(!solved, "problem has been solved already")
      low to midpoint

   def solved: Boolean = low == high

   def yes(): Unit =
      requireState(!solved, "problem has been solved already")
      high = midpoint
```

```
def no(): Unit =
   requireState(!solved, "problem has been solved already")
   low = midpoint + 1

def progress: Double =
   import scala.math.log
   if solved then 1.0 else 1 - log(high - low + 1) / log(size)

def secret: Int =
   requireState(solved, "problem not solved yet")
   low
```

There are a few noticeable differences in syntax:

- A `def` keyword is used to introduce each method. It is followed by the method's name, its list of arguments and their types, then by the return type of the method. The return type is sometimes left unspecified, to be inferred by the compiler, as in method `midpoint`.
- The keyword `def` can be preceded by a visibility modifier. Contrary to Java, the default semantics (no modifier) means *public*—while in Java, it means private to the package. There is no `public` keyword in Scala.
- All methods have a return type. Methods that were `void` in Java use the return type `Unit` in Scala. This type contains only one (useless) value and is appropriate for methods that modify an object but do not return anything meaningful (like `void` methods in Java). Sections 3.1 and 3.2 discuss the notion of pure/impure functions and the use of `Unit` as return type. There is no `void` keyword in Scala.
- Methods that take no arguments can be defined without an empty pair of parentheses. Methods with no arguments that only query the state of an object are usually defined without parentheses, since calling them is no different from reading a field (this is often referred to as the *uniform access principle*).[1] By contrast, methods that modify the state of the object typically keep the parentheses, whether they need arguments or not. In the case of `HiLo`, only the methods `yes` and `no` (which do indeed modify the state of the object) use parentheses.
- The body of a method is an expression. The value returned by a method is the value of this expression. No `return` keyword is used. Some expressions are simple, like `(low + high) / 2` or `low == high`. More complex expressions can be built using blocks. Blocks are delimited by curly braces, which can sometimes be left out and inferred by the compiler (as in the code above). The value of a block expression is the value of the last expression in the block.

[1]The Scala standard library tends to stretch the uniform access principle a bit. For instance, the `iterator` method of collections is defined without parentheses, even though it is not equivalent to reading a field, since every invocation needs to allocate a new iterator. Still, it is a method that does not modify the state of its target object.

For instance, method `secret` uses a block expression. The first expression in the block is used to check the state of the game (and to potentially throw an exception). The second expression, `low`, is the value returned by the method. In Scala, there are no statements, only expressions—a fundamental principle of functional programming (see Section 3.3). Blocks can include many types of declarations, such as local variables and classes (like in Java), but also functions (see `yesOrNo` and `getReply` below) or import statements (as in method `progress`).

- The last expression evaluated in the body of method `progress` is an `if-then-else`. In Scala, `if-then-else` is an expression, not a statement, and it produces a value that can be used as a return value or as an argument for a method call. Accordingly, both Java's `if-else` and the ternary operator `<cond>?<expr1>:<expr2>` can be replaced with `if-then-else` in Scala.[2]

The remainder of the code, which is *static* in Java, is found in Scala in the companion object, named `HiLo`, like the class:

Scala

```scala
object HiLo:
   def upto(max: Int): HiLo = HiLo(1, max)

   private def play(hilo: HiLo) =
      def yesOrNo(line: String): Option[Boolean] =
         line.trim match
            case "y" | "Y" | "yes" | "Yes" | "YES" => Some(true)
            case "n" | "N" | "no" | "No" | "NO"    => Some(false)
            case _                                 => None

      def getReply(question: String): Boolean =
         yesOrNo(scala.io.StdIn.readLine(question)) match
            case Some(ans) => ans
            case None      => getReply("  yes or no? ")

      println(s"Playing HiLo between ${hilo.low} and ${hilo.high}.")

      while !hilo.solved do
         val choices = hilo.choices
         val question =
            if choices.sizeIs == 1 then s"Is your number ${choices.head} ? "
            else s"Is your number between ${choices.head} and ${choices.last} ? "
         if getReply(question) then hilo.yes() else hilo.no()
         println(f"I'm ${hilo.progress * 100.0}%.0f%% done.")

      println(s"Your number is: ${hilo.secret}.")
   end play
```

[2]Scala 2 used an `if (<cond>) <expr1> else <expr2>` syntax. This syntax is still available in Scala 3, although `if <cond> then <expr1> else <expr2>` is now preferred.

Method `upto` creates a new instance of the game. It could have been written: `new HiLo(1, max)`. The use of keyword `new` is optional here in Scala.

Method `play`, which implements the terminal-based interaction with the user, defines a helper function `yesOrNo` similar to the Java method by the same name. One difference is that `yesOrNo` is defined inside `play`, as a local function. Another difference with Java is that the use of `switch` has been replaced with pattern matching, a feature that is similar but much more powerful. (Pattern matching is discussed in Chapter 5.)

A more substantial difference is that the inner `while` loop used in Java's `play` method is replaced here by a call to a helper function `getReply`. Like the loop in Java, this function is used to query the user until a valid response is entered. Function `getReply` uses recursion instead of a loop, but is actually compiled into a loop by the Scala compiler. (See the discussion of tail-recursive functions in Section 6.5.) Recursion, including tail recursion, is discussed at length in Chapters 6 and 7.

Because function `play` is somewhat lengthy, an "`end play`" marker is used to improve legibility. End markers can be used to terminate the definition of a variable, function, or class. They are optional.

A final difference between the two *HiLo* implementations is the use of string interpolation in Scala instead of the `formatted` method in Java. In Scala, string interpolation is triggered by a leading `s` or `f` in front of a string literal:

Scala
```scala
s"Pi is not ${22.0 / 7.0}"      // "Pi is not 3.142857142857143"
f"Pi is not ${22.0 / 7.0}%.2f" // "Pi is not 3.14"
```

Finally, the main program is implemented in a `main` method of the companion object, with a signature similar to that of a static `main` method in Java:

Scala
```scala
  ...
  def main(args: Array[String]): Unit =
    def usage() = println("Usage: HiLo <max>")

    if args.length != 1 then usage()
    else args(0).toIntOption match
      case Some(max) if max >= 1 => play(HiLo.upto(max))
      case _                     => usage()
end HiLo
```

Another local function is used for `usage`, and the parsing of a string into an integer relies on options and pattern matching instead of catching exceptions in the Java variant.

B.4 A Few Scala Idiosyncrasies

Scala is a fairly modern language and was carefully crafted. However, no programming language is entirely free of features that will seem surprising to programmers as they transition into the language. This section discusses several peculiarities of Scala that a Java developers may find confusing at first.

Hidden methods

Several methods play a special role in Scala, based on their names. Consider, for instance, this code that uses a map from project identities to programming languages:

Scala

```scala
val language: Map[Int, String] = Map(54321 -> "Scala", 12345 -> "C")

language(54321)          // "Scala"
language(12345) = "Rust" // sets 12345 language to "Rust"
```

The lookup in the map, `language(54321)`, is actually a call to a method `apply` of the map object. It could have been written: `language.apply(54321)`. Similarly, the update to the map is a call to a method `update` and could have been written: `language.update(12345, "Rust")`. Even the creation of the map is a call to a method `apply` of the `Map` companion object.

For this reason, array indexing in Scala uses parentheses instead of Java's square brackets: The expression `x = array(n)` is `x = array.apply(n)`, and `array(n) = x` is `array.update(n, x)`.

Two other methods, `unapply` and `unapplySeq`, play a special role in implementing pattern matching extractors (see Section 5.7).

No break or continue

Scala has no `break` or `continue` statement. Scala's programming style often favors recursion over iteration. Many `while` loops that would be written using `break` or `continue` in Java are written as recursive functions in Scala and can dispense with the `break` and `continue` statements altogether:

Java

```java
void processFirst(Elements source) {
  while (source.hasNext()) {
    Element element = source.next();
    if (element.hasDesiredProperty()) {
      process(element);
      break;
    }
  }
}
```

This Java method extracts elements from a source and processes the first element, if any, that satisfies a desired property. If no element satisfies the property, the method does nothing. The `break` keyword is used to ensure that only the first element with the property is processed.

In Scala, this method could be written recursively instead:

Scala
```scala
def processFirst(source: Elements): Unit =
   if source.hasNext then
      val element = source.next()
      if element.hasDesiredProperty then process(element) else processFirst(source)
```

The method is tail recursive and will be compiled as a loop (see discussion of tail recursion in Section 6.5). At the bytecode level, it will be very similar to the Java version.[3]

Implicit arguments

Scala uses a notion of arguments that are passed to functions implicitly:

Scala
```scala
val languages = List("Fortran", "C",  "BASIC", "COBOL", "Ada", "Java", "Bash")
languages.sorted // List(Ada, BASIC, Bash, C, COBOL, Fortran, Java)
```

Method `sorted` takes an implicit argument or type `Ordering`. The Scala standard library defines a default value that compares strings according to their natural order. As a consequence, `BASIC` appears before `Bash`, because `A` comes before `a` in ASCII order. This could be changed by defining a different implicit ordering, for instance one that compares strings in a case-insensitive way:

Scala
```scala
given Ordering[String] = (x: String, y: String) => x.compareToIgnoreCase(y)
```

With this definition is scope, the same call `languages.sorted` has `Bash` appear before `BASIC`.

Single-argument methods called on block expressions

In the discussion of the Scala implementation of *HiLo*, I mentioned that blocks are expressions, with a value. Therefore, a block can be used as an argument when calling a method, as in:

[3]For this particular method, however, an experienced Scala programmer would likely avoid both looping and recursion, and rely on higher-order methods instead:

```scala
def processFirst(source: Elements): Unit =
   for (element <- source.find(_.hasDesiredProperty)) do process(element)
```

Higher-order methods and Scala's `for-do` are discussed in Chapter 9.

Scala

```scala
val out: PrintWriter = ...

out.println({
   val two = 2
   two + two
})
```

The value on which `println` is called is clearly delimited by the curly braces of the block, and the parentheses are redundant. Scala allows programmers to omit the unnecessary parentheses and to write instead:

Scala

```scala
out.println {
   val two = 2
   two + two
}
```

In Scala 3.2, used in the book, this is only possible on blocks delimited by curly braces, not indentation. Starting with 3.3, Scala makes it possible to invoke methods on blocks delimited by indentation:

Scala

```scala
out.println:
   val two = 2
   two + two
```

Omitting the parentheses may seem like a small thing, but it impacts how code is written and can be puzzling to programmers new to the language. For instance, we will see in Chapter 25 how to schedule asynchronous tasks on a thread pool using a construct of the form:

Scala

```scala
val f = Future {
   // some code
}
```

What is actually compiled here is a call to a hidden method `apply` on a block of code (omitting extraneous parentheses), using an implicit argument for a thread pool. It could have been written:

Scala

```scala
val f = Future.apply({
   // some code
})(using summon[ExecutionContext])
```