PROGRAMACIÓN CONCURRENTE Y AVANZADA

CUADERNO DE PRÁCTICAS

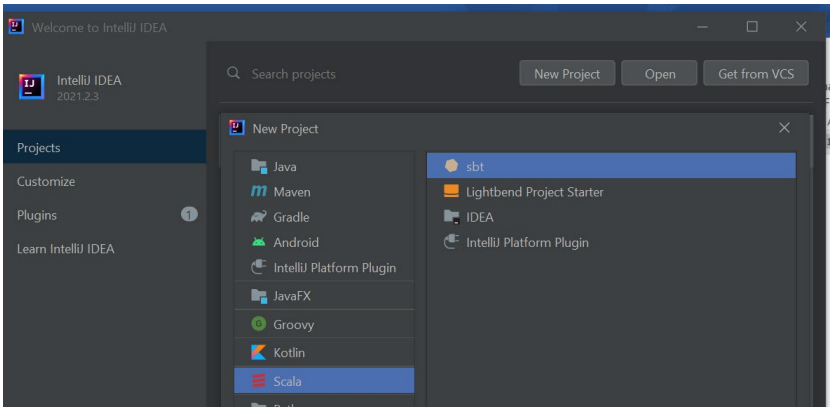
P02. Scala

Vamos a familiarizarnos con la introducción de código Scala en el entorno **IntelliJ** a través de un programa muy sencillo:

```
object helloScala {
  def main(args: Array[String]) {
     println ("Bienvenidos a Scala")
  }
}
```

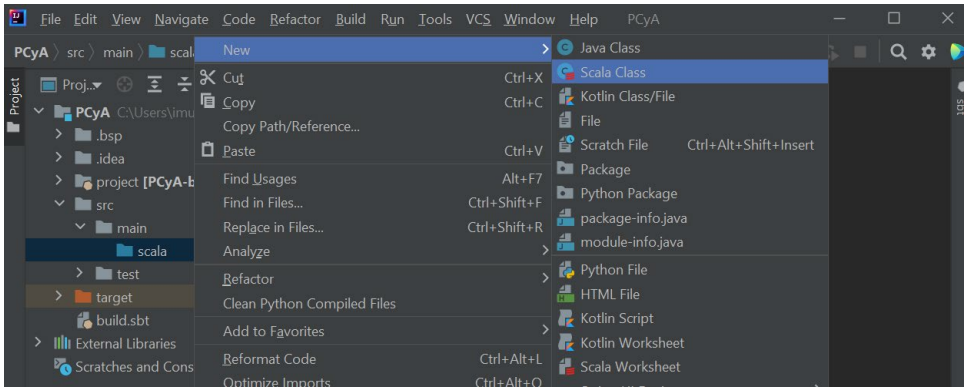
Para ello, arranca el entorno IntelliJ y sigue los siguientes pasos:

- Crea un proyecto a través de los siguientes pasos:
  `File -> New Project -> Scala -> sbt`
  pulsa Next y en la siguiente pantalla indica el nombre del proyecto, por ejemplo, `PCyA`



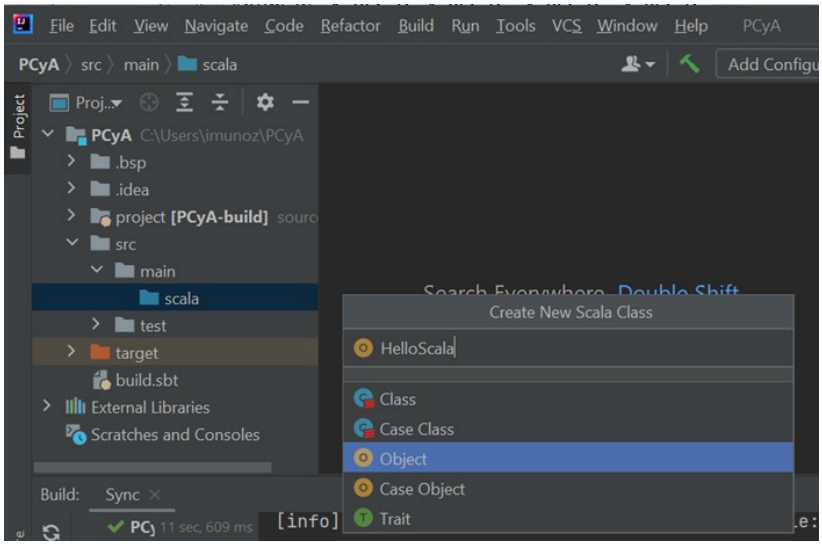
El entorno crea el proyecto como se indica en la figura (en este caso con el nombre PCyA) con carpeta `src` (entre otras) desde donde colgarán los objetos que se creen.

- Pincha sobre `PCyA -> src -> main -> scala`, con el botón derecho del ratón:



- Selecciona `New -> Scala Class`.

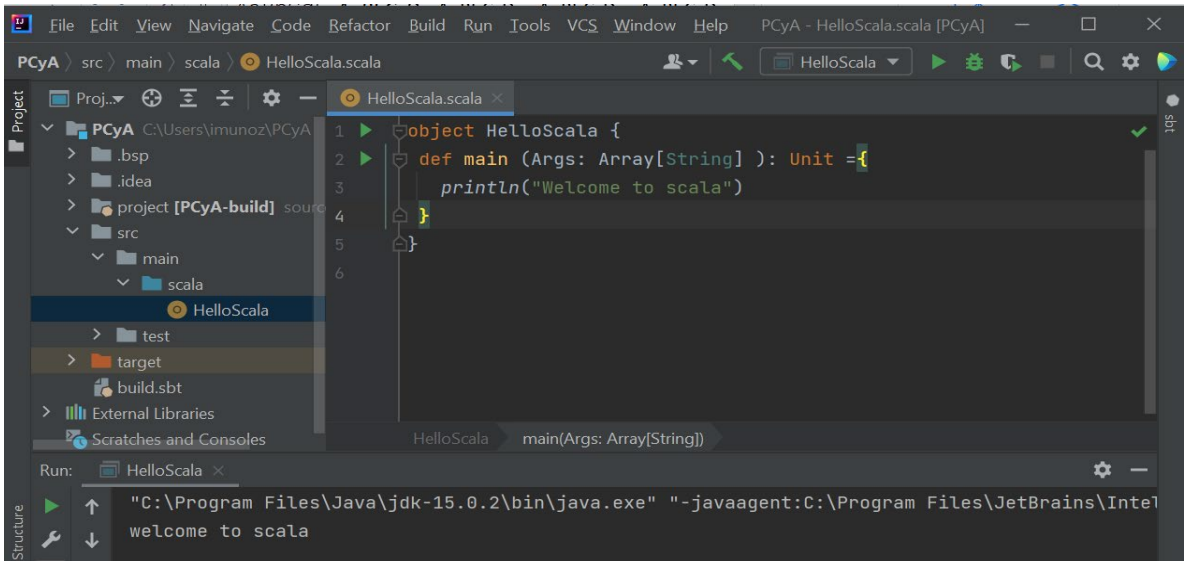
Verás un nuevo menu “Create New Scala Class” con differentes clases de Scala. Selecciona la opcion “Object” y denominalo “HelloScala”.



En la ventana de código del entorno, aparecerá el esqueleto del nuevo objeto.

- Complétalo para obtener el programa mostrado arriba. Una vez completo, salva su contenido.

Puedes apreciar, en la figura siguiente, que el sistema ha creado el fichero `helloScala.scala` y lo ha situado debajo del paquete scala `default package` (que se creó de manera automática al no haber especificado un paquete)



A continuación, ejecútalo (Menu `Run -> Run… -> HelloScala`). O bien haciéndo click en el triángulo verde junto al método main. Comprueba que en la consola de salida muestra la salida esperada.

En realidad, para que se ejecute el procedimiento `println` habría que importar una biblioteca (o librería) disponible en el lenguaje, pero esto se hace automáticamente con algunas bibliotecas.

Scala importa implícitamente los paquetes `java.lang`, `scala` y `Predef`. Algunos ejemplos de objetos importados implícitamente son:

```
Predef.println
java.lang.String
scala.Int
scala.Char
scala.Double
```

Gracias a la importación implícita de los paquetes citados, podemos usar estos objetos sin necesidad de indicar de dónde provienen. Así, el tipo entero se nombrará `Int` en vez de `scala.Int`, sin indicar explícitamente qué paquete lo contiene.

1. [AU] En la sección "Funciones" del Tutorial de Scala que has revisado con el entorno Kojo, había una función `max` para calcular el máximo de dos números. Copia al entorno de Scala el siguiente programa. Selecciona en este caso File -> `New -> Scala class -> Object`, pon como nombre `pru` y completa con el siguiente código:

```
object pru extends App {
  def max(x: Int, y: Int): Int = {
    if (x > y) x
    else y
  }
  println (max(3,4))
}
```

Ejecútalo y comprueba el resultado.

A continuación, se proporciona la definición de la clase `Point` del apartado "Objects and Classes" de Kojo. Además, se proporciona un programa que usa dicha clase. Crea un nuevo objeto llamado PPoint y copia el programa anterior al entorno de Scala.

```
case class Point(x:Int,y:Int){
  def +(newpt:Point)=Point(x+newpt.x,y+newpt.y)
  def -(newpt:Point)=Point(x-newpt.x,y-newpt.y)
  override def toString="Point("+x+","+y+")"
}

object PPoint {
  def main(args: Array[String]) {
    val p1=Point(3,4)
    val p2=Point(7,2)
    val p3=Point(-2,2)
    val p4=p1+p2-p3
    println(p4)
  }
}
```

Indica qué hace el programa y por qué se imprime el contenido de `p4` sin ser un `String`. Si no sabes la respuesta, lee la sección titulada "Case classes" al final de este guion. De hecho, deberías leer la sección titulada "El lenguaje Scala" antes de proseguir.

Indica cómo es posible que el programa anterior no dé errores de compilación siendo que no están definidos en la clase `Point` las variables `x` e `y`, y sus métodos de acceso. Prueba a quitar el modificador `case` de la clase `Point` y a introducir el modificador `val` en cada uno de sus parámetros, así como eliminar (basta con comentarlo al principio de la línea) la sobreescritura del método `toString`. Ahora la creación de objetos debe hacerse con el método `new`. ¿Qué se imprime ahora?

2. [AU] Introduce y ejecuta el siguiente programa:

```
object Ejemplo2 extends App {
```

```
  def obtenerSuma(args: Int *) : Int = {
    var suma : Int = 0
    for (num <- args) {
      suma += num
    }
    suma
  }
  println ("Suma obtenida " + obtenerSuma(1,2,3,4,5,6))
}
```

```
Suma obtenida 21
```

Este mismo resultado puede obtenerse utilizando el operador `fold (/:)`

```
def obtenerSuma(args: Int *) : Int = {
      args.foldLeft(0) (_+_)
}
```

Pruébalo en el programa anterior e intenta explicar el funcionamiento del operador. Para ello sería conveniente leer la sección "Folding lists" al final de este guion.

3. [AU] Scala permite multiplicar un string por un número. Por ejemplo `"loco" * 3`. ¿Cuál es el resultado? ¿Puedes explicarlo? Mira la siguiente URL:

https://www.scala-lang.org/api/2.12.3/scala/collection/immutable/StringOps.html

4. [AU] Escribe una expresión en Scala equivalente a la siguiente expresión en Java:

```
for (int i = 10; i >= 0; i--) System.out.println(i);
```

5. [LA] Escribe una función que calcule x^n, donde n es un entero. Usa la siguiente definición recursiva:

- $x^n = y \cdot y$, si n es par y positivo, donde $y = x^{n/2}$.
- $x^n = x \cdot x^{n-1}$, si n is impar y positivo.
- $x^0 = 1$.
- $x^n = 1 / (x^{-n})$, si n es negativo.

No uses la sentencia `return`. Utiliza la sentencia `match`. Sigue el siguiente esquema:

```
object Ejemplo3 extends App {
  def potencia (x: Double, n: Int): Double = {
    n match {
      case 0 => 1
      case n: Int if (n < 0) => …
      …
    }
  }
  println (" 4 elevado a -1: ", potencia (4, -1))
  println (" 4 elevado a 0: ", potencia (4, 0))
  println (" 4 elevado a 3: ", potencia (4, 3))
  println (" 4 elevado a 2: ", potencia (4, 2))
}
```

6. [LA] Elimina todos los elementos negativos de un Array de enteros. Sigue el siguiente esquema:

```
object Ejemplo4 extends App {
  def eliminaNegativos (a: Array[Int]) : Array[Int] = {
    ……
  }
  val b = Array (2, -1, 3, 4, -5)
  val res = eliminaNegativos (b)
```

```
  // imprime el resultado
}
```

Pista: recuerda que la sentencia `for` puede llevar un filtro asociado y el modificador `yield`.

7. [LA] Dado un array de números enteros, genera un nuevo array que contenga todos los valores positivos del array original, en su orden original, seguidos por todos los valores que son cero o negativos, en su orden original. Sugerencia: utiliza la primitiva *partition* o *filter* desde la consola de Scala en IntelliJ (https://www.scala-lang.org/api/current/scala/Array.html)

```
scala> val c = Array (0, 1, 2, -3, -4)
c: Array[Int] = Array(0, 1, 2, -3, -4)
-----------------------------------------------------------
```

8. [LA] Escribe una función `lteqgt(values: Array[Int], v: Int)` que devuelve un triplete (Tuple3) conteniendo el número de elementos del array `values` que son menores que `v`, igual a `v` y mayor que `v`.

```
object Ejemplo5 extends App {
  def lteqgt(values: Array[Int], v: Int): (Int, Int, Int) = {
    ……
  }
  val valores = Array (2, -1, 3, 4, -5)
  val valor = 3
  val res = lteqgt(valores, valor)
  println (res.toString)
}
```

Case classes

Crea una nueva Scala App llamada `Ejemplo1` e introduce el siguiente código

```
object Ejemplo1 extends App {
  val v = Var ("x")
  val op = BinOp ("+", Number(1), v)
  println (v.name)
  println (op.left)
  println (op)
}
```

El sistema indica errores porque faltan declarar algunos tipos. Introduce (antes o después del código anterior, es indiferente) las siguientes declaraciones:

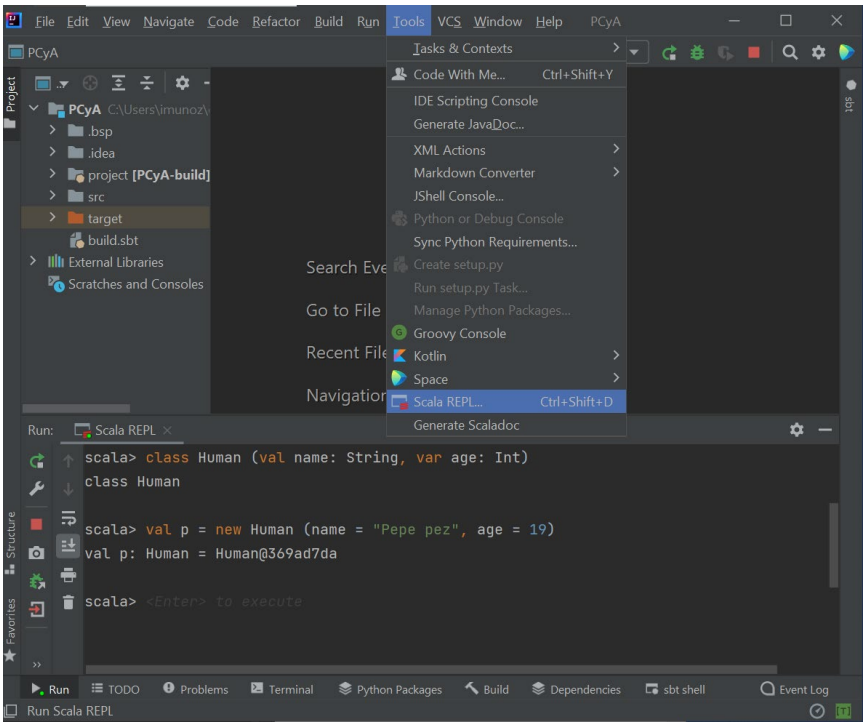
```
abstract class Expr
case class Var (name: String) extends Expr
case class Number (num: Double) extends Expr
case class UnOp (operator: String, arg: Expr) extends Expr
case class BinOp (operator: String, left: Expr, right: Expr) extends Expr
```

Si ejecutas las expresiones anteriores en el entorno de Scala, la salida obtenida es:

```
x
Number(1.0)
BinOp(+,Number(1.0),Var(x))
```

En las expresiones anteriores puedes observar que cada subclase tiene el modificador `case`. Las clases con tal modificador se denominan *case classes*. El uso de tal modificador instruye al compilador para que haga algún trabajo por nosotros.

En primer lugar, añade un método Factory con el nombre de la clase. Esto significa que, por ejemplo, puedes escribir `Var("x")` para construir un objeto de tipo `Var` en lugar de utilizar la expresión: `new Var("x")`. Abre la `Scala Console` tal y como se indica en la siguiente figura.



Evalua la siguiente expresión

```
scala> val v = Var("x")
v: Var = Var(x)
```

En segundo lugar, el compilador añade el prefijo `val` a todos los argumentos en la lista de parámetros de la *case class*. Esto tiene el efecto de añadir un atributo a la *case class* con el mismo nombre que el parámetro y que puede ser accedido desde fuera de la clase.

```
scala> v.name
res0: String = x
```

Como norma general para los parámetros de los constructores primarios se siguen las siguientes reglas:

Parámetro constructor primario	Campos/métodos generados
`nombre: String`	campo privado al objeto, o sin ningún campo si ningún método usa `nombre`
`private val/var nombre: String`	Campo privado; `getter/setter` privado
`val/var nombre: String`	Campo privado; `getter/setter` público

En tercer lugar, el compilador añade una implementación de los métodos `toString`, `hasCode` y `equals` a la *case class*. El método toString de la clase Object devuelve un string que consiste en el nombre de la clase seguido del character @ y la representación hexadecimal del código hash del objeto. El método `toString` que añade el compilador escribirá un árbol que consiste en el nombre de la clase y, recursivamente, el valor de todos sus argumentos.

Folding lists

La operación *fold left* "(z /: xs) (op)" implica tres objetos: un valor inicial `z`, una lista `xs` y una operación `op`. El resultado de *fold* es el operador binario `op` aplicado entre elementos sucesivos de la lista `xs`, comenzando con el valor `z`. Por ejemplo:

```
scala> List(1,2,3).foldLeft(10)(_*_)
val res1: Int = 60
```

En este ejemplo, se se coge el valor inicial, 10, y el primer elemento de la lista, 1, y se multiplican dando como resultado 10. Este valor ahora se considera como valor inicial y se multiplica por el segundo elemento de la lista, 2, resultando en 20. De nuevo, se considera 20 como valor inicial y ahora se multiplica por el tercer y último elemento de la lista, 3, resultando en el valor final de 60. Observa que el valor inicial actúa como un contenedor donde se van almacenado los resultados parciales.

El lenguaje Scala

Aprende a utilizar el intérprete de Scala

La forma más fácil de empezar con Scala es utilizar el intérprete de Scala, una "shell" interactiva, para escribir expresiones y programas en Scala. El intérprete, llamado `scala`, evalúa las expresiones que escribes e imprime el valor resultante. Para ejecutar el intérprete: menu `Tools -> ScalaRPL` para ver el prompt:

```
scala>
```

Si no se ve el prompt "`scala>`" en la consola de la parte de abajo de la pantalla, debe abrirse la "Scala Console" a través del tercer icono desde la derecha en la zona de la consola (parte de debajo de la pantalla).

Cuando introduces una expresión, por ejemplo `1 + 2`, y tecleas enter:

```
scala> 1 + 2
```

el intérprete responderá:

```
res0: Int = 3
```

Esta línea incluye:

- un nombre generado automáticamente o definido por el usuario para referirse al valor calculado (`res0`, que significa resultado 0),
- un símbolo ":", seguido por el tipo de la expresión (`Int`),
- un signo "=",
- el valor resultante de evaluar la expresión (`3`).

El tipo `Int` nombra la clase `Int` del paquete `scala`. Los paquetes en Scala son similares a los de Java: dividen el espacio de nombres global y proporcionan un mecanismo para ocultar información. Los valores de la clase `Int` corresponden a los valores `int` de Java. De forma más general, todos los tipos primitivos de Java tienen sus clases correspondientes en el paquete `scala`. Por ejemplo, `scala.Boolean` corresponde a `Boolean` de Java; `scala.Float` corresponde a `Float` de Java. Cuando se traduce del código Scala a Java bytecodes, el compilador de Scala usará los tipos primitivos de Java cuando sea posible para proporcionar las ventajas de los tipos primitivos.

El identificador `resX` puede usarse posteriormente. Por ejemplo, como `res0` se inicializó a `3` previamente, `res0 * 3` será `9`:

```
scala> res0 * 3
res1: Int = 9
```

Para imprimir el saludo `Hello, world!`:

```
scala> println("Hello, world!")
Hello, world!
```

La función `println` imprime el string pasado en la salida estándar, de forma similar a `System.out.println` en Java.

Define algunas variables

Scala tiene dos tipos de variables, `vals` y `vars`. Una variable `val` es similar a una variable final en Java. Una vez inicializado, un `val` no puede reasignarse. Un `var`, por el contrario, es similar a una variable no final en Java. Un `var` puede reasignarse a lo largo de su vida. Aquí vemos una definición de `val`:

```
scala> val msg = "Hello, world!"
msg: String = Hello, world!
```

Esta sentencia asigna `msg` como nombre para el string `"Hello, world!"`. El tipo de `msg` es `java.lang.String`, ya que los strings de Scala se implementan a través de la clase `String` de Java.

Si estás acostumbrado a declarar variables en Java, notarás una diferencia chocante aquí: ni `java.lang.String` ni `String` aparecen en ninguna parte de la definición de `val`. Este ejemplo ilustra la inferencia de tipos (*type inference)*, la habilidad de Scala para adivinar tipos que no se especifican. En este caso, dado que se inicializó `msg` con un string literal, Scala infiere que el tipo de `msg` es `String`. Cuando el intérprete (o compilador) de Scala puede inferir tipos, es mejor dejar que lo haga en vez de introducir código con anotaciones explícitas innecesarias. Sin embargo, puedes especificar explícitamente un tipo si quieres, y a veces deberías. Una anotación explícita de tipo puede asegurar que el compilador de Scala infiere el tipo que se pretende, y puede servir como documentación para futuros lectores del código. A diferencia de Java, donde se especifica el tipo de la variable antes de su nombre, en Scala se especifica el tipo de la variable después de su nombre, separado por ":". Por ejemplo:

```
scala> val msg2: java.lang.String = "Hello again, world!"
msg2: String = Hello again, world!
```

Además, como los tipos de `java.lang` son visibles con sus nombres simples en Scala, puede escribirse:

```
scala> val msg3: String = "Hello yet again, world!"
msg3: String = Hello yet again, world!
```

Volviendo al `msg` original, ahora que está definido, puedes usarlo como era de esperar:

```
scala> println(msg)
Hello, world!
```

Lo que no puedes hacer con `msg`, dado que es un `val`, y no un `var`, es reasignarlo. Por ejemplo, puedes ver cómo protesta el intérprete si ejecutas lo siguiente:

```
scala> msg = "Goodbye cruel world!"
<console>:8: error: reassignment to val
       msg = "Goodbye cruel world!"
           ^
```

Si quieres reasignar, necesitarás usar un `var`:

```
scala> var greeting = "Hello, world!"
greeting: String = Hello, world!
```

Como `greeting` es un `var` y no un `val`, puedes reasignarlo más tarde. Si estás enfadado, por ejemplo, podrías cambiar `greeting`:

```
scala> greeting = "Leave me alone, world!"
greeting: String = Leave me alone, world!
```

Para introducir más de una línea en el intérprete, sigue tecleando tras la primera línea. Si el código tecleado por el momento no está completo, el intérprete responderá con una barra vertical en la siguiente línea.

```
scala> val multiLine =
     |   "This is the next line."
multiLine: String = This is the next line.
```

Si ves que has tecleado algo erróneo, pero el intérprete aún está esperando más entrada, puedes anularlo tecleando enter dos veces:

```
scala> val oops =
     |
     |
You typed two blank lines.  Starting a new command.
scala>
```

Define algunas funciones

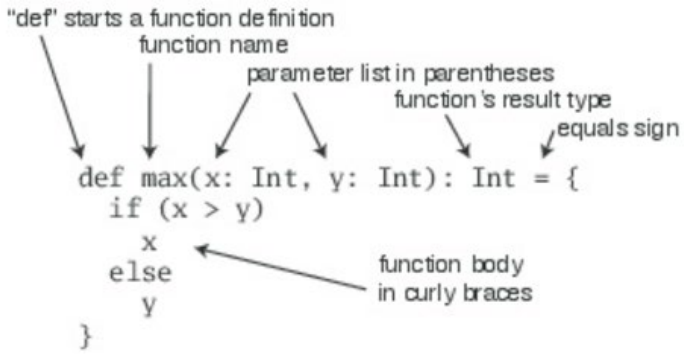
Ahora que has trabajado con variables en Scala, probablemente querrás escribir algunas funciones. Así se hace en Scala:

```
scala> def max(x: Int, y: Int): Int = {
         if (x > y) x
         else y
       }
```

Y el interprete responde con:

```
max: (x: Int, y: Int)Int
```

Las definiciones de función empiezan con `def`. Tras el nombre de la función, en este caso `max`, se especifica la lista de parámetros separados por comas entre paréntesis. Cada parámetro debe ir seguido por una anotación de tipo, precedido por ":", ya que el compilador de Scala (y el intérprete, aunque a partir de ahora solo diremos compilador) no infiere los tipos de los parámetros. En este ejemplo, la función llamada `max` requiere dos parámetros, `x` e `y`, ambos de tipo `Int`. Tras el paréntesis cerrado de los parámetros de `max`, aparece otra anotación de tipo "`: Int`". Esta define el *result type* de la función `max`. Detrás del tipo de la función, hay un signo "=" y unas llaves que contienen el cuerpo de la función. En este caso, el cuerpo contiene solo una expresión `if`, que selecciona el mayor entre `x` e `y` como resultado de la función `max`. Como se ha mostrado aquí, la expression `if` de Scala puede resultar en un valor, de forma similar al operador ternario de Java. Por ejemplo, la expression de Scala "`if (x > y) x else y`" tiene un comportamiento similar a "`(x > y)? x : y`" en Java. El signo igual que precede al cuerpo de la función da pistas de la visión funcional, donde una función define una expresión que resulta en un valor. La estructura básica de una función es:



Algunas veces el compilador de Scala requiere que se especifique el resultado de una función. Por ejemplo, si la función es *recursiva*, se debe especificar explícitamente el tipo del resultado de la función. En el caso de `max`, sin embargo, puedes omitir el tipo del resultado y el compilador podrá inferirlo. También, si una función solo tiene una sentencia, pueden eliminarse las llaves. Así, una forma alternativa de escribir la función `max` sería:

```
scala> def max(x: Int, y: Int) = if (x > y) x else y
max: (x: Int, y: Int)Int
```

Una vez definida la función, puedes llamarla por su nombre:

```
scala> max(3, 5)
res4: Int = 5
```

A continuación, puedes ver una función que no lleva parámetros y no devuelve ningún resultado de interés:

```
scala> def greet() = println("Hello, world!")
greet: ()Unit
```

Cuando se define la función `greet()`, el intérprete responderá `greet: ()Unit`. "`greet`" es el nombre de la función. Los paréntesis vacíos indican que la función no requiere parámetros. `Unit` es el tipo de resultado de `greet`. El tipo de resultado `Unit` indica que la función no devuelve ningún resultado de interés. El tipo `Unit` de Scala es similar al `void` de Java. Los métodos que devuelven `Unit`, por tanto, se ejecutan solo por los efectos laterales. En el caso de `greet()`, el efecto lateral es un saludo por la salida estándar.

Iteración con foreach y for

En el estilo imperativo, normal en lenguajes como Java, C++, y C, se proporciona una orden cada vez, se itera con bucles, y a menudo se comparte un estado mutable entre diferentes funciones. Scala permite programación imperativa, pero cuando lo conozcas mejor, probablemente programarás en un estilo más funcional. Una de las principales características de un lenguaje funcional es que las funciones son constructors de primera clase, y esto es cierto en Scala. Por ejemplo, una forma bastante más concisa de imprimir cada parámetro en la línea de commandos es:

```
args.foreach(arg => println(arg))
```

En este código, llamas al método `foreach` con `args` y pasas una función. En este caso, se pasa un *function literal* que toma un parámetro llamado `arg`. El cuerpo de la función es `println(arg)`. Si `args` es "Concise is nice", se visualizaría:

```
Concise
is
nice
```

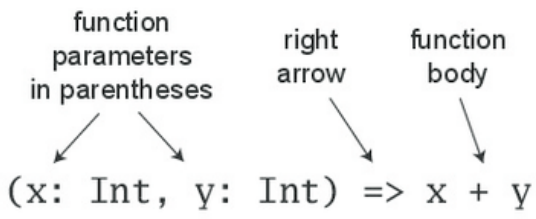
En el ejemplo anterior, el intérprete de Scala infiere que el tipo de `arg` es `String`, ya que `String` es el tipo de element del array con que se llama a `foreach`. Si prefieres ser más explícito, puedes mencionar el nombre del tipo. Pero si lo haces, tendrás que rodear la parte de parámetros entre paréntesis:

```
args.foreach((arg: String) => println(arg))
```

Si quieres mayor concision, puedes usar un atajo de Scala. Si una función literal consiste en una sentencia que lleva un único parámetro, no necesitas nombrarlo y especificarlo. Así, el siguiente código también funcionaría:

```
args.foreach(println)
```

Resumiendo, la sintaxis de una función literal es una lista de parámetros con nombre, entre paréntesis, una flecha, y el cuerpo de la función:



En este punto puedes estar preguntándote qué pasó con aquellos bucles a los que estás acostumbrado en lenguajes imperativos como Java o C. En un esfuerzo por guiarte en el estilo funcional, en Scala solo encontrarás un pariente funcional del for imperative (llamado *for expression*). Teclea lo siguiente en un nuevo fichero llamado `forargs.scala`:

```
for (arg <- args)
  println(arg)
```

El paréntesis tras el "`for`" contiene `arg <- args`. A la derecha del símbolo `<-` está el familiar array `args`. A la izquierda de `<-` está "`arg`". Para cada elemento del array `args`, se creará un *nuevo* `arg`, se inicializará con el valor del elemento, y se ejecutará el cuerpo del for.

Si ejecutas el script `forargs.scala` con el comando:

```
$ scala forargs.scala for arg in args
```

verás:

```
for
arg
in
args
```

Singleton objects

Una forma en que Scala es más orientado a objetos que Java es que las clases de Scala no pueden tener miembros estáticos. En vez de ello, Scala tiene *singleton objects*. Una definición de singleton object se parece a una definición de clase, excepto que en vez de la palabra reservada `class` se usa `object`. Listing 4.2 muestra un ejemplo.

```
// In file ChecksumAccumulator.scala

  import scala.collection.mutable
    object ChecksumAccumulator {
    private val cache = mutable.Map.empty[String, Int]
    def calculate(s: String): Int =
      if (cache.contains(s))
        cache(s)
      else {
        val acc = new ChecksumAccumulator
        for (c <- s)
          acc.add(c.toByte)
        val cs = acc.checksum()
        cache += (s -> cs)
        cs
      }
  }
```

Listing 4.2 – Objeto Companion para la clase ChecksumAccumulator.

El singleton object se llama `ChecksumAccumulator`, igual que la clase.

```
class ChecksumAccumulator {
  private var sum = 0
  def add (b: Byte): Unit = {sum += b}
  def checksum (): Int = ~(sum & 0xFF) + 1
}
```

Cuando un singleton object comparte el mismo nombre que una clase, se denomina *companion object* de esa clase. Debe definirse la clase y su companion object en el mismo fichero fuente. La clase se denomina *companion class* del singleton object. Una clase y su companion object pueden acceder mútuamente a sus miembros privados.

El `ChecksumAccumulator` singleton object tiene un método llamado `calculate`, que dado un `String` calcula un checksum para los caracteres del `String`. Tiene también un campo privado, `cache`, un mutable map en que se almacenan los checksums calculados previamente. La primera línea del método, "`if (cache.contains(s))`", comprueba la cache para ver si el string pasado ya está contenido como clave en el mapa. Si es así, devuelve el mapped value, `cache(s)`. Si no, ejecuta la parte else, que calcula el checksum. La primera línea de la parte else define un `val` llamado `acc` y lo inicializa con una instancia de `ChecksumAccumulator`. La siguiente línea es una expresión `for` que se ejecuta con cada carácter del string pasado, convierte el carácter a un `Byte` llamando a `toByte`, y se lo pasa al método `add` de la instancia de `ChecksumAccumulator` referenciadas por `acc`. Cuando se completa la expresión `for`, la siguiente línea del método invoca a `checksum` con `acc`, obteniendo el checksum del `String` pasado y lo almacena en un `val` llamado `cs`. En la siguiente línea, `cache += (s -> cs)`, el string key pasado se mapea al valor entero de checksum, y el par key-value se añade al cache `map`. La última expresión del método, `cs`, asegura que el checksum es el resultado del método.

Si eres programador de Java, puedes pensar en los singleton objects como contenedores para métodos estáticos escritos en Java. Puedes llamar a los métodos de los singleton objects con una sintaxis similar: el nombre del singleton object, un punto y el nombre del método. Por ejemplo, puedes llamar al método `calculate` del singleton object `ChecksumAccumulator` como sigue:

```
ChecksumAccumulator.calculate("Every value is an object.")
```

Una diferencia entre clases y singleton objects es que los singleton objects no pueden llevar parámetros, mientras que las clases sí. Como no puedes instanciar un singleton object con `new`, no tienes una forma para pasarle parámetros. Un singleton object que no comparte el mismo nombre que una companion class se denomina *standalone object*. Puedes usar standalone objects para muchas cosas, como agrupar utility methods o definir un punto de entrada a una aplicación Scala.

Una aplicación Scala

Para ejecutar un programa Scala, debes proporcionar el nombre de un standalone singleton object con un método `main` que lleva un parámetro, un `Array[String]`, y tiene un resultado de tipo `Unit`. Cualquier objeto standalone con un método `main` puede usarse como punto de entrada a una aplicación. Se muestra un ejemplo en Listing 4.3:

```
// In file Summer.scala
import ChecksumAccumulator.calculate
object Summer {
  def main(args: Array[String]) = {
    for (arg <- args)
      println(arg + ": " + calculate(arg))
  }
}
```

Listing 4.3 – Aplicación Summer.

El nombre del singleton object en Listing 4.3 es `Summer`. Su método `main` tiene la sintaxis adecuada y puede usarse como una aplicación. La primera sentencia del fichero es un import del método `calculate` definido en el objeto `ChecksumAccumulator` del ejemplo anterior. Esta sentencia import permite usar el nombre simple del método en el resto del fichero. El cuerpo del método `main` imprime cada argumento y su checksum correspondiente, separados por dos puntos.

Nota: Scala importa implícitamente los miembros de los paquetes `java.lang` y `scala`, así como los miembros del singleton object llamado `Predef`, en todos los ficheros fuente de Scala. `Predef`, que reside en el paquete `scala`, contiene muchos métodos de utilidad. Por ejemplo,

cuando dices `println` en un fichero fuente de Scala, estás llamando al `println` de `Predef`. (`Predef.println` llama a `Console.println`, que hace el trabajo).

Para ejecutar la aplicación `Summer`, incluye el código de Listing 4.3 en un fichero llamado `Summer.scala`. Como `Summer` usa `ChecksumAccumulator`, incluye el código de `ChecksumAccumulator`, tanto la clase como su companion object.

```
import scala.collection.mutable
object ChecksumAccumulator {
  private val cache = mutable.Map.empty[String, Int]
  def calculate(s: String): Int =
    if (cache.contains(s))
      cache(s)
    else {
      val acc = new ChecksumAccumulator
      for (c <- s)
        acc.add(c.toByte)
      val cs = acc.checksum()
      cache += (s -> cs)
      cs
    }
}

class ChecksumAccumulator {
  private var sum = 0
  def add(b: Byte): Unit = { sum += b }
  def checksum(): Int = ~(sum & 0xFF) + 1
}
```

Una diferencia entre Scala y Java es que mientras Java requiere que una public class se ponga en un fichero llamado como la clase—por ejemplo, pondrías la clase `SpeedRacer` en el fichero `SpeedRacer.java`—en Scala puedes llamar a los ficheros `.scala` como quieras, independientemente de las clases Scala o el código que pongas en ellos. Sin embargo, en el caso de non-scripts se recomienda llamar a los ficheros como las clases que contienen como en Java, de forma que los programadores puedan localizar las clases más fácilmente mirando los nombres de los ficheros. Este es el método que hemos seguido con los dos ficheros de este ejemplo, `Summer.scala` y `ChecksumAccumulator.scala`.

Puedes ejecutar la aplicación `Summer` con los parámetros (`of love`): `Run>Run configurations…>(x)Arguments` y escribir, en la caja `Program arguments`, las palabras "of love" y pulsar `Run`.

The App trait

Scala proporciona un trait, `scala.App`, que puede ahorrarte teclear. El listado 4.4 muestra un ejemplo:

```
import ChecksumAccumulator.calculate
object FallWinterSpringSummer extends App {
  for (season <- List("fall", "winter", "spring"))
    println(season + ": " + calculate(season))
}
```

Listing 4.4 – Uso del `App` trait.

Para usar el trait, escribes `extends App` tras el nombre de tu singleton object. Después, en lugar de escribir un método `main`, pones el código que habrías puesto en el método `main` directamente entre las llaves del singleton object. Puedes acceder a los parámetros mediante un array de strings llamado `args`. Puedes compilar y ejecutar esta aplicación como cualquier otra.

Tipos básicos

En la Tabla 5.1 se muestran algunos tipos fundamentales de Scala, junto a los rangos de valores de estos tipos. Conjuntamente, los tipos `Byte`, `Short`, `Int`, `Long` y `Char` se denominan *integral types*. Los integral types más `Float` y `Double` se llaman *numeric types*.

Table 5.1 - Some basic types

Basic type	Range
Byte	8-bit signed two's complement integer (-2^7 to $2^7 - 1$, inclusive)
Short	16-bit signed two's complement integer (-2^{15} to $2^{15} - 1$, inclusive)
Int	32-bit signed two's complement integer (-2^{31} to $2^{31} - 1$, inclusive)
Long	64-bit signed two's complement integer (-2^{63} to $2^{63} - 1$, inclusive)
Char	16-bit unsigned Unicode character (0 to $2^{16} - 1$, inclusive)
String	a sequence of Chars
Float	32-bit IEEE 754 single-precision float
Double	64-bit IEEE 754 double-precision float
Boolean	true or false

Excepto `String`, que reside en el paquete `java.lang`, todos los tipos mostrados en la Tabla 5.1 son miembros del paquete `scala`. Por ejemplo, el nombre completo de `Int` es `scala.Int`. Sin embargo, dado que todos los miembros del paquete `scala` y `java.lang` se importan automáticamente en todos los ficheros fuente de Scala, puedes usar los nombres simples (*por ejemplo*, nombres como `Boolean`, `Char`, o `String`) en cualquier sitio.

Literales

Todos los tipos básicos listados en Table 5.1 pueden escribirse con *literals*. Un literal es una forma de escribir un valor constante directamente en el código.

Literales Integer

Los literales Integer para los tipos `Int`, `Long`, `Short` y `Byte` vienen de dos formas: decimal y hexadecimal. La forma en que comienza el literal indica la base del número. Si el número empieza con `0x` o `0X`, es hexadecimal (base 16), y puede contener dígitos de `0` a `9` y dígitos `A` a `F` en mayúsculas o minúsculas:

```
scala> val hex = 0x5
hex: Int = 5

scala> val hex2 = 0x00FF
hex2: Int = 255

scala> val magic = 0xcafebabe
magic: Int = -889275714
```

Nota que la shell de Scala siempre escribe los valores integer en base 10, independientemente de la forma de literal que se haya usado para inicializarlos. Así, el intérprete visualiza el valor de la variable `hex2` inicializada con el literal `0x00FF` como el decimal `255`. Una buena forma de empezar a familiarizarse con el lenguaje es probar estas sentencias en el intérprete según lees el capítulo. Si el número empieza con un dígito distinto de cero y no tiene ninguna otra decoración, es decimal (base 10). Por ejemplo:

```
scala> val dec1 = 31
dec1: Int = 31

scala> val dec2 = 255
dec2: Int = 255
```

```
scala> val dec3 = 20
dec3: Int = 20
```

Si un literal integer acaba con `L` o `l`, es un `Long`; si no, es `Int`. Algunos ejemplos de literales `Long` integer son:

```
scala> val prog = 0XCAFEBABEL
prog: Long = 3405691582

scala> val tower = 35L
tower: Long = 35

scala> val of = 31l
of: Long = 31
```

Si un literal `Int` se asigna a una variable de tipo `Short` o `Byte`, el literal se trata como si fuera de tipo `Short` o `Byte` siempre que el valor del literal esté dentro del rango válido para este tipo. Por ejemplo:

```
scala> val little: Short = 367
little: Short = 367

scala> val littler: Byte = 38
littler: Byte = 38
```

Literales Floating point

Los literales floating point están formados por dígitos decimales, opcionalmente seguidos por un punto decimal y dígitos decimales, y opcionalmente seguido por `E` o `e` y un exponente. Algunos ejemplos de literales floating-point son:

```
scala> val big = 1.2345
big: Double = 1.2345

scala> val bigger = 1.2345e1
bigger: Double = 12.345

scala> val biggerStill = 123E45
biggerStill: Double = 1.23E47
```

Nota que la parte de exponente indica la potencia de 10 a la que se eleva la otra porción. Así, `1.2345e1` es `1.2345` *times* 10^1, que es `12.345`. Si un literal floating-point termina en `F` o `f`, es un `Float`; si no es un `Double`. Optionalmente, un `Double` floating-point literal puede terminar en `D` o `d`. Algunos ejemplos de literales Float son:

```
scala> val little = 1.2345F
little: Float = 1.2345

scala> val littleBigger = 3e5f
littleBigger: Float = 300000.0
```

Ese último valor expresado como `Double` podría tomar otras formas:

```
scala> val anotherDouble = 3e5
anotherDouble: Double = 300000.0

scala> val yetAnother = 3e5D
yetAnother: Double = 300000.0
```

Literales Character

Los literales Character se componen de cualquier carácter Unicode entre comillas simples, tales como:

```
scala> val a = 'A'
a: Char = A
```

Además de proporcionar un carácter explícito entre comillas simples, un carácter puede identificarse usando su código Unicode. Para ello, escribe \u seguido por 4 dígitos hexadecimales con su código, como sigue:

```
scala> val d = '\u0041'
d: Char = A

scala> val f = '\u0044'
f: Char = D
```

De hecho, estos caracteres Unicode pueden aparecer en cualquier sitio en un programa Scala. Por ejemplo, podrías escribir un identificador como éste:

```
scala> val B\u0041\u0044 = 1
BAD: Int = 1
```

Este identificador se trata igual que `BAD`, resultado de expandir los dos caracteres Unicode en el código anterior. En general es mala idea nombrar así a los identificadores porque es difícil de leer. Esta sintaxis está pensada para permitir en Scala introducir caracteres Unicode no ASCII en un fichero ASCII.

Finalmente, hay también unos pocos literales carácter representados por secuencias especiales, que se muestran en la Tabla 5.2. Por ejemplo:

```
scala> val backslash = '\\'
backslash: Char = \
```

Table 5.2 – Secuencias especiales de literales character

Literal	Meaning
\n	line feed (\u000A)
\b	backspace (\u0008)
\t	tab (\u0009)
\f	form feed (\u000C)
\r	carriage return (\u000D)
\"	double quote (\u0022)
\'	single quote (\u0027)
\\	backslash (\u005C)

Literales String

Un literal string está compuesto de caracteres rodeados por comillas dobles:

```
scala> val hello = "hello"
hello: String = hello
```

La sintaxis de los caracteres entre comillas es la misma que en literales character. Por ejemplo:

```
scala> val escapes = "\\\"\'"
escapes: String = \"'
```

Como esta sintaxis es muy rara para strings que contienen varias secuencias escape o strings que ocupan varias líneas, Scala incluye una sintaxis especial para *raw strings*. Un raw string se empieza y se acaba con 3 comillas dobles consecutivas ("""). El interior de un raw string puede contener cualquier carácter, incluyendo saltos de línea, comillas y caracteres especiales, excepto 3 comillas dobles consecutivas. Por ejemplo, el siguiente programa imprime un mensaje usando un raw string:

```
println("""Welcome to Ultamix 3000.
           Type "HELP" for help.""")
```

Sin embargo, ejecutar este código no produce el resultado deseado:

```
Welcome to Ultamix 3000.
           Type "HELP" for help.
```

El problema es que los espacios que preceden a la segunda línea se incluyen en el string. Para resolver este problema, puedes llamar a `stripMargin` para strings. Para usar este método, pon un carácter pipe (`|`) al comienzo de cada línea, y llama a `stripMargin` para el string completo:

```
println("""|Welcome to Ultamix 3000.
           |Type "HELP" for help.""".stripMargin)
```

Ahora el código se comporta como queríamos:

```
Welcome to Ultamix 3000.
Type "HELP" for help.
```

Literales Boolean

El tipo `Boolean` tiene dos literales, `true` y `false`:

```
scala> val bool = true
bool: Boolean = true

scala> val fool = false
fool: Boolean = false
```

String interpolation

Scala incluye un mecanismo flexible para string interpolation, que permite empotrar expresiones dentro de literales string. Su uso más común es proporcionar una alternativa concisa y legible a la concatenación de strings. Este es un ejemplo:

```
val name = "reader"
println(s"Hello, $name!")
```

La expresión `s"Hello, $name!"` es un *processed* string literal. Como la letra `s` precede inmediatamente a las comillas abiertas, Scala usará el *`s` string interpolator* para procesar el literal. El `s` interpolator evaluará cada expresión empotrada, llamando a `toString` para cada resultado, y reemplazará las expresiones empotradas en el literal por sus resultados. Así, `s"Hello, $name!"` genera `"Hello, reader!"`, el mismo resultado que `"Hello, " + name + "!"`.

Puedes situar cualquier expresión después de un dollar sign (`$`) en un processed string literal. Para expresiones de una sola variable, puedes poner el nombre de la variable tras el dollar sign. Scala interpretará todos los caracteres hasta el primer carácter non-identifier como la expresión. Si la expresión incluye caracteres non-identifier, debes incluirla entre llaves, con la llave inicial siguiendo inmediatamente al dollar sign. Este es un ejemplo:

```
scala> s"The answer is ${6 * 7}."
res0: String = The answer is 42.
```

Scala proporciona otros dos string interpolators por defecto: `raw` y `f`. El `raw` string interpolator se comporta como `s`, excepto que no reconoce literal escape sequences (tales como las mostradas en Table 5.2).

Table 5.2 - Special character literal escape sequences

Literal	Meaning
\n	line feed (\u000A)
\b	backspace (\u0008)
\t	tab (\u0009)
\f	form feed (\u000C)
\r	carriage return (\u000D)
\"	double quote (\u0022)

\'	single quote (\u0027)
\\	backslash (\u005C)

Por ejemplo, la sentencia siguiente imprime cuatro backslashes, no dos:

```
println(raw"No\\\\escape!") // prints: No\\\\escape!
```

El `f` string interpolator permite añadir formateo del tipo `printf` a expresiones. Se sitúan las instrucciones después de la expresión, comenzando por el símbolo de porcentaje (%), usando la sintaxis especificada por `java.util.Formatter`. Por ejemplo, aquí se muestra cómo mostrar el número pi con formato:

```
scala> f"${math.Pi}%.5f"
res1: String = 3.14159
```

Si no proporcionas instrucciones de formato en la expresión, el `f` string interpolator tendrá el formato `%s` por defecto, lo que significa que se sustituirá por el valor `toString`, de igual forma que el `s` string interpolator. Por ejemplo:

```
scala> val pi = "Pi"
pi: String = Pi

scala> f"$pi is approximately ${math.Pi}%.8f."
res2: String = Pi is approximately 3.14159265.
```

Expresión For

La expresión `for` de Scala es la navaja suiza de la iteración. Permite combinar unos pocos ingredientes de manera diferente para expresar una amplia variedad de iteraciones. Algunos usos sencillos permiten tareas comunes como iterar sobre una secuencia de enteros, mientras que expresiones más avanzadas pueden iterar sobre colecciones múltiples de distintos tipos, filtrar elementos basados en condiciones arbitrarias y producir nuevas colecciones.

Iteración sobre collections

Lo más simple que puedes hacer con un `for` es iterar sobre los elementos de una collection. Por ejemplo, Listing 7.5 muestra código para imprimir todos los ficheros del directorio actual. La E/S se realiza usando la API de Java. Primero creamos un `java.io.File` en el directorio actual, ".", y llamamos al método `listFiles`. Este método devuelve un array de objetos `File`, uno por cada directorio y fichero contenido en el directorio actual. El array resultante se almacena en la variable `filesHere`.

```
val filesHere = (new java.io.File(".")).listFiles
for (file <- filesHere) println(file)
```

Listing 7.5 – Listado de ficheros en un directorio con una expresión for.

Con la sintaxis "`file <- filesHere`", que se denomina *generator*, iteramos sobre los elementos de `filesHere`. En cada iteración, se inicializa un nuevo `val` llamado `file` con uno de los elementos. El compilador infiere que el tipo de `file` es `File`, porque `filesHere` es un `Array[File]`. Por cada iteración, se ejecuta el cuerpo de la expresión `for`, `println(file)`. Como el método `toString` de `File` devuelve el nombre del fichero o directorio, se imprimirán los nombres de todos los ficheros y directorios del directorio actual.

La sintaxis de la expresión `for` funciona para cualquier tipo de colección, no solo arrays. Un caso especial es el tipo `Range`. Puedes crear `Range`s usando la sintaxis "`1 to 5`" y puedes iterar sobre ellos con un `for`. Por ejemplo:

```
scala> for (i <- 1 to 4)
         println("Iteration " + i)
```

```
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

Si no quieres incluir el límite superior entre los valores a iterar, puedes usar `until` en vez de `to`:

```
scala> for (i <- 1 until 4)
         println("Iteration " + i)
Iteration 1
Iteration 2
Iteration 3
```

La iteración sobre enteros es muy común en Scala, pero no tanto como en otros lenguajes. En otros lenguajes, puedes usarlo para iterar sobre los elementos de un array, como sigue:

```
// Not common in Scala...
for (i <- 0 to filesHere.length - 1)
  println(filesHere(i))
```

Esta expresión `for` introduce una variable `i` que va tomando todos los valores entre `0` y `filesHere.length - 1`, y ejecuta el cuerpo de la expresión `for` para cada valor de `i`. Es decir, para cada valor que toma `i`, se extrae y procesa el i-ésimo element de `filesHere`.

La razón para que este tipo de iteración sea menos común en Scala es que puedes iterar directamente sobre la colección. Cuando lo haces, el código resultante es más corto y puedes evitar muchos de los errores que pueden aparecer al iterar sobre arrays. ¿Debes empezar con 0 o con 1? ¿Debes sumar -1, +1 o nada al índice final? Estas preguntas se contestan fácilmente, pero también es fácil cometer errores al contestarlas. Por ello, es más seguro evitarlas.

Filtering

Algunas veces no queremos iterar sobre una colección completa, sino que queremos filtrar algún subconjunto. Esto se puede hacer con una expresión `for` añadiendo un *filter*, que consiste en una sentencia `if` dentro de los paréntesis del `for`. Por ejemplo, el código mostrado en Listing 7.6 lista solo los ficheros del directorio con nombre acabado en `".scala"`:

```
val filesHere = (new java.io.File(".")).listFiles
for (file <- filesHere if file.getName.endsWith(".scala"))
  println(file)
```

Listing 7.6 – Búsqueda de ficheros .scala usando un for con un filtro.

También se podría conseguir el mismo objetivo con este código:

```
for (file <- filesHere)
  if (file.getName.endsWith(".scala"))
    println(file)
```

Este código genera la misma salida que el anterior, y probablemente resulte más familiar a los programadores con experiencia en lenguajes imperativos. La forma imperativa, sin embargo, es solo una opción porque esta expresión `for` se ejecuta por su efecto lateral de imprimir y devuelve Unit (). Como se demuestra más tarde en esta sección, la expresión `for` se considera "expresión" porque puede dar como resultado un valor interesante, una colección cuyo tipo se determina por las sentencias <- de la expresión `for`.

Pueden incluirse más filtros incluyendo más sentencias `if`. Por ejemplo, poniéndose muy defensivo, el código de Listing 7.7 imprime solo ficheros y no directorios. Esto lo hace añadiendo un filtro que lo comprueba con el método `isFile` de `file`.

```
for (
  file <- filesHere
  if file.isFile
```

```
    if file.getName.endsWith(".scala")
  ) println(file)
```

Listing 7.7 – Uso de varios filtros en una expresión for.

Iteración anidada

Si incluyes multiples sentencias <- habrá bucles anidados. Por ejemplo, la expresión `for` mostrada en Listing 7.8 tiene dos bucles anidados. El bucle exterior itera sobre `filesHere`, y el interior itera sobre `fileLines(file)` para cualquier `file` que acabe con `.scala`.

```
def fileLines(file: java.io.File) =
  scala.io.Source.fromFile(file).getLines().toList

def grep(pattern: String) =
  for (
    file <- filesHere
    if file.getName.endsWith(".scala");
    line <- fileLines(file)
    if line.trim.matches(pattern)
  ) println(file + ": " + line.trim)

grep(".*gcd.*")
```

Listing 7.8 – Uso de multiple generators en una expresión `for`.

Mid-stream variable bindings

Note que el código anterior repite la expresión `line.trim`. Se trata de un proceso no trivial, por lo que puede ser deseable hacerlo solo una vez. Esto puede hacerse dejando el resultado en una nueva variable mediante el signo igual (=). La variable ligada se introduce y se usa como un `val`, únicamente excluyendo la palabra reservada `val`. Listing 7.9 muestra un ejemplo.

```
def grep(pattern: String) =
  for {
    file <- filesHere
    if file.getName.endsWith(".scala")
    line <- fileLines(file)
    trimmed = line.trim
    if trimmed.matches(pattern)
  } println(file + ": " + trimmed)

grep(".*gcd.*")
```

Listing 7.9 - Mid-stream assignment en una expresión for.

En Listing 7.9, se introduce en mitad de la expresión `for` una variable llamada `trimmed`. Esa variable se inicializa con el resultado de `line.trim`. El resto de la expresión `for` usa la nueva variable en dos sitios, una vez en el `if` y otra en el `println`.

Creación de una nueva colección (collection)

Mientras todos los ejemplos anteriores operaban con las variables iteradas y después las olvidaban, también se puede generar un valor en cada iteración del `for` y recordarlo. Para ello, se introduce la palabra `yield` antes del cuerpo de la expresión `for`. Por ejemplo, aquí se proporciona una función que identifica los ficheros `.scala` y los almacena en un array:

```
def scalaFiles =
  for {
    file <- filesHere
    if file.getName.endsWith(".scala")
  } yield file
```

Cada vez que se ejecuta el cuerpo de la expresión `for`, se produce un valor, en este caso, `file`. Cuando se complete la expresión `for`, el resultado incluirá todos los valores generados por la función en una colección única. El tipo de colección resultante depende de las colecciones procesadas en las sentencias de iteración. En este caso, el resultado es un `Array[File]`, ya que `filesHere` es un array y el tipo de expresión generada es `File`.

Debes tener cuidado con el lugar donde sitúas la palabra `yield`. La sintaxis de una expresión `for-yield` es la siguiente:

```
for clauses yield body
```

La palabra `yield` va antes del cuerpo de la expresión `for`. Incluso si el cuerpo es un bloque entre llaves, `yield` debe ir antes de la llave. Debe evitarse la tentación de escribir algo como esto:

```
for (file <- filesHere if file.getName.endsWith(".scala")) {
  yield file  // Syntax error!
}
```

Implicit imports

Scala añade algunos imports implícitamente a todos los programas. Básicamente, es como si se añadiera al principio de todos los programas las siguientes sentencias:

```
import java.lang._ // everything in the java.lang package
import scala._     // everything in the scala package
import Predef._    // everything in the Predef object
```

El paquete `java.lang` contiene standard Java classes. Siempre se importa implícitamente en ficheros fuente de Scala. Como `java.lang` se importa implícitamente, puedes escribir `Thread` en vez de `java.lang.Thread`, por ejemplo.

Como probablemente te habrás dado cuenta, el paquete `scala` contiene la biblioteca estándar de Scala, con muchas clases y objetos muy comunes. Como `scala` se importa implícitamente, puedes escribir `List` en vez de `scala.List`, por ejemplo.

El objeto `Predef` contiene muchas definiciones de tipos, métodos y conversiones implícitas que se usan mucho en programas Scala. Por ejemplo, como `Predef` se importa implícitamente, puedes escribir `assert` en vez de `Predef.assert`.

Estas tres sentencias import se tratan de forma especial, ya que los imports posteriores se superponen a los anteriores. Por ejemplo, la clase `StringBuilder` está definida tanto en el paquete `scala` como en Java (version 1.5 y siguientes), en el paquete `java.lang`. Como la importación de `scala` se superpone a la de `java.lang`, el nombre simple `StringBuilder` hará referencia a `scala.StringBuilder`, y no a `java.lang.StringBuilder`.

Assertions

Las aserciones en Scala se escriben como llamadas al método predefinido `assert`. La expresión `assert(condition)` provoca un `AssertionError` si `condition` no se cumple. También hay una versión de `assert` con dos parámetros. La expresión `assert(condition, explanation)` evalúa `condition` y, si no se cumple, provoca un `AssertionError` que contiene la explicación. El tipo de explicación es `Any`, por lo que puedes pasar cualquier objeto como explicación. El método `assert` llamará a `toString` para poner el string `explanation` dentro de `AssertionError`.