

Chapter 10

Scala Methods



10.1 Introduction

This chapter presents how methods and associated behaviour is defined in Scala.

10.2 Method Definitions

Methods provide a way of defining the behaviour of an object, i.e. what the object does. For example, a method may change the state of the object or it may retrieve some information. A method is the equivalent of a procedure in most other languages. A method can only be defined within the scope of an object. It has a specific structure:

```
access-control-modifier  
def methodName (args: argTypes): returnType = {  
    /* comments */  
    local variable definitions  
    statements  
}
```

The *access control modifier* is one of the keywords that indicate the visibility of the method. The `returnType` is the type of the object returned; for example, `String`, or `Int`. `methodName` represents the name of the method and `args` represents the types and names of the arguments. These arguments are accessible within the method:

```

object MathUtil {
  def max(x: Int, y: Int): Int = {
    if (x > y)
      return x
    else
      return y
  }
}

```

The above definition defines a method ‘max’ as part of an object MathUtil. The method takes two parameters both of type Int, one accessible via the parameter x and the other via the parameter y. The return type of the method is explicitly specified to be Int. Both of the methods contain an ‘if’ conditional statement which will return x if x is greater than y, otherwise it will return y.

Scala is quiet flexible in the way that methods are defined. The above could also be rewritten as:

```

object MathUtil {
  def max2(x: Int, y: Int) = if (x>y) x else y
}

```

Both max and max2 do exactly the same thing and seen by Scala as exactly the same (as Scala infers much of what has been omitted).

In the following example class the different variations on defining the method *greet* are all valid.

```

object BasicMethodTest {

  def greet(): Unit = {
    println("Hello World!")
  }

  def greet2() = {
    println("Hello World!")
  }

  def greet3() = println("Hello World")

  def greet4() { println("Hello World!") }

  def greet5 = println("Hello World")

}

```

The methods are discussed below:

- The `greet()` method. This is a longhand definition of a method. It has a set of parentheses indicating that no parameters are required. It explicitly states that `Unit` (nothing) is the return type. And the body of the method follows the `'='` symbol surrounded by curly brackets.
- The `greet2()` method. This version defaults the return type to be `Unit`; this is the default if nothing is stated as the return type of `println` is also `Unit`.
- The `greet3()` method. This method defaults the return type and does not include `{}` as it is a single line definition.
- The `greet4()` method. This includes brackets around the statements, but defaults the return type and does not include the `'='`—this is known as *procedural* style.
- The `greet5` method does not even include the `()` as they are not needed.

Invoking methods `greet` to `greet4` can be invoked with or without `'()'`. However, it should be noted that `greet5` can only be involved without parameters, for example,

```
object TestMethods extends App {
  BasicMethodTest.greet()
  BasicMethodTest.greet

  BasicMethodTest.greet2()
  BasicMethodTest.greet2

  BasicMethodTest.greet3()
  BasicMethodTest.greet3

  BasicMethodTest.greet4()
  BasicMethodTest.greet4

  // BasicMethodTest.greet5() - invalid
  BasicMethodTest.greet5
}
```

10.2.1 Method Parameters

Methods can take parameters. Parameters are values or instances that are passed into a functional unit such as a method. If the parameters are reference types (i.e. instances) then a copy of the reference is passed in. However, as the reference is essentially the address of the underlying object in memory, this means that a parameter refers to the same instance as any values that hold that reference externally to the method.

In Scala a method can take Zero or more parameters. The `main` method that you have seen several times in the last chapter takes a single parameter of type `Array[String]`. That is, it holds a reference to an array of strings. This is shown in the following example.

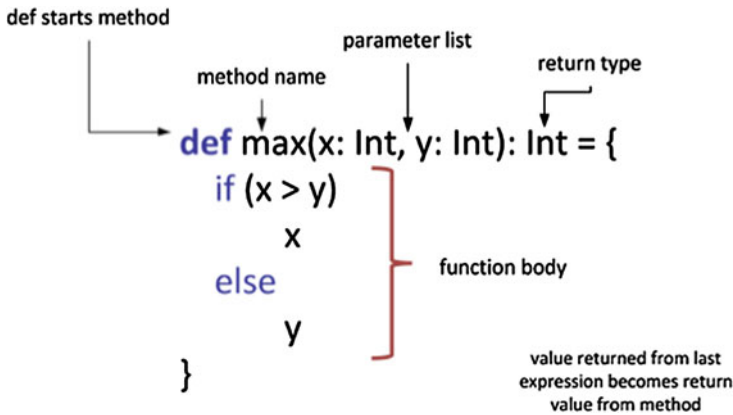


Fig. 10.1 Elements of a method

```

object MethodTest {
  def main(args: Array[String]): Unit = {
    println(max(2, 3))
  }

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

```

In this example, as well as the main method, a second method has also been defined within the `MethodTest` object. This method is called `max` and takes two parameters. That is, there are two parameters in the parameter list for the method, in this case `x` and `y`. The anatomy of the method is explored in more detail in Fig. 10.1.

The keyword `def` started the declaration. Here the method name is `max`, it has two parameters in its parameter list (of type `Int`) called `x` and `y`, and it returns an `Int`. Within the `{..}` is the method body which in this case considers `x` relative to `y` and returns (by default) either `x` or `y`.

Within the method all parameters are *vals*; that is it is not possible to assign a new value to a parameter to the method. That is, you cannot reassign a value to `x` or `y` in the above example. Thus it is not possible to write:

```

def add(i: Int): Unit {
  i = i + 1 // won't compile i is a val
}

```

10.2.2 Comments

The `/*` comments `*/` section describes the operation performed by the method and any other useful information. Comments cannot be nested in Scala, which can be awkward if you wish to comment out some code for later. For example, consider the following piece of code:

```
/*
val x = 12 * 4
/* Now calculate y */
val y = x * 23
*/
```

The Scala compiler reads this as a comment, followed by the code `y = x * 23`; followed by the end of another comment. This causes an error. However, Scala has two other types of comment. You can instruct the Scala compiler to ignore everything until the end of the line, using the `//` indicator:

```
val x = 12 * 4
// Now calculate y
val y = x * 23
```

The final type of comment, the documentation comment, starts with `/**` and ends with `*/`. Note the two asterisks at the beginning of this statement. They are picked up and processed by the documentation utility (scaladoc), which generates HTML pages that can be viewed in a Web browser. They can contain wiki markup and other control directives. These directives are defined as `@<directive>`, for example,

`@constructor`—used to provide documentation for the constructor

`@param`—used to provide documentation for a parameter

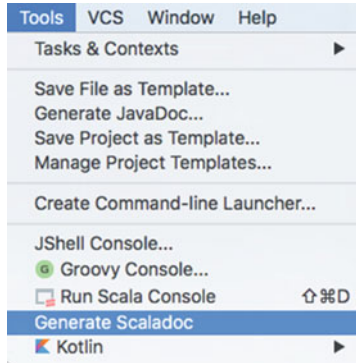
`@return`—used to provide documentation on a return type.

An example of such Scaladoc comments is shown below:

```
package com.jjh.scala.person

/**
 * A person who uses our application.
 *
 * @constructor create a new person with a name and age.
 * @param name the person's name
 * @param age the person's age in years
 */
class Person(val name: String, var age: Int)
```

In IntelliJ it is possible to run the Scaladoc command from within the tool. On the main menu bar, see the ‘Tools->Generate Scaladoc’ option:



This will present you with a dialog allowing you to control what you want to apply the scaladoc command to.

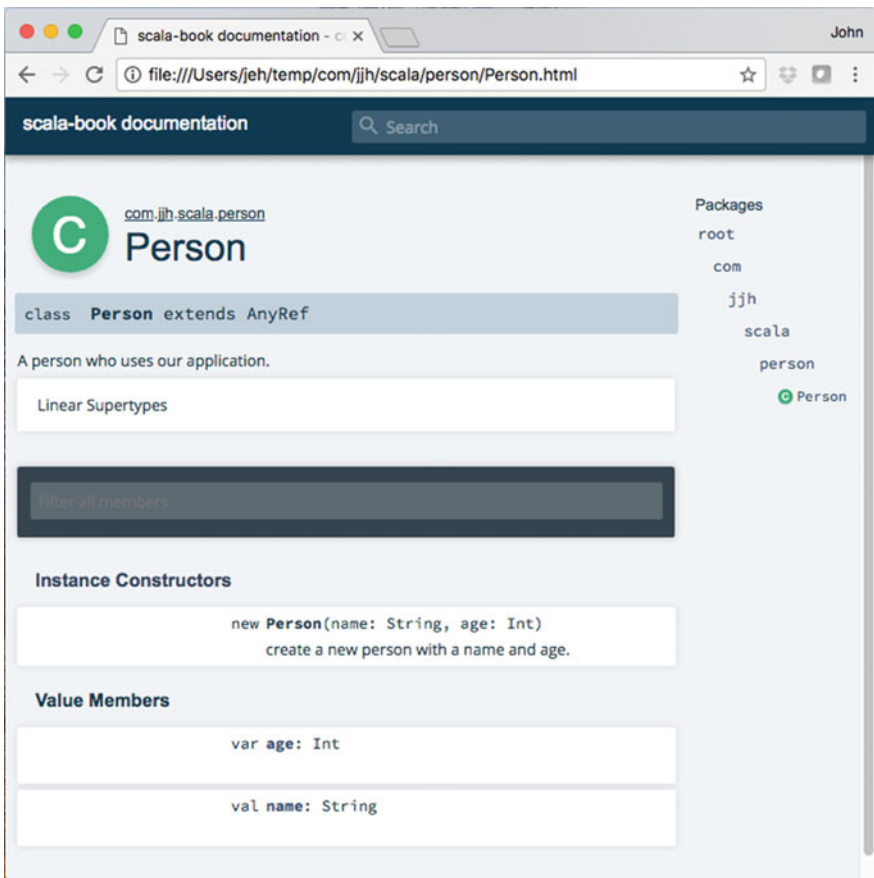


Fig. 10.2 Scaladoc-generated reference material for the Person class

The result of running the scaladoc tool against this class is shown in Fig. 10.2.

In fact the whole of the reference material available for Scala has been produced using Scaladoc. A more complex example of which is shown below. This presents the scaladoc for the Scala class AnyRef.

The screenshot shows the Scala Standard Library API documentation for the `AnyVal` class. The browser address bar shows `www.scala-lang.org/api/current/scala/AnyVal.html`. The page title is "Scala Standard Library 2.12.4". The main content area features the Scala logo and the text "abstract class AnyVal extends Any". Below this, there is a detailed description of `AnyVal` as the root class of all value types, followed by a list of subtypes and a minimal example code snippet:

```
class Wrapper(val underlying: Int) extends AnyVal {
  def foo: Wrapper = new Wrapper(underlying * 19)
}
```

On the right side, there is a "Packages" sidebar listing various Scala packages, including `scala`, `annotation`, `beans`, `collection`, `compat`, `concurrent`, `io`, `math`, `ref`, `reflect`, `runtime`, `sys`, `text`, `util`, and `Any`.

This Scaladoc is available online at

- <http://www.scala-lang.org/api/current>

10.2.3 The Local Variables Section

In the local variable definition section, you define variables which are local to the method. These variables are typed and can appear anywhere in the method

definition. They are only available within the method definition itself and have no meaning elsewhere and are not visible elsewhere.

```
birthday ()
  val newAge = 0;
  ...
```

The variables may be vals or vars depending on whether you want to allow reassignment to them or not. However, it is worth noting that by convention in Scala, vals are preferred and many IDEs will mark vars in red as a warning that they should not be used.

10.2.4 The Statements Section

The statements section represents any legal set of Scala statements that implement the behaviour of the method.

10.2.5 The Return Operator

Once a method has executed, an answer can be returned to the sender of the message. The value returned (whether an object, a basic type or an instance of a subclass) must match the return type specified (or inferred by Scala) in the method definition. The return expression in Scala is the last expression executed in a method, although it need not be the last expression in the method.

The Scala keyword to return a value is `return` (just as in Java); however, it is optional as the result of the last expression will automatically be returned if the method returns something other than `Unit`; thus, the following are equivalent:

```
if (x == y)
  return x;
else
  return y;
```

Or

```
if (x == y)
  x;
else
  y;
```


In both these cases, the value of `x` or `y` is returned, depending upon whether `x` and `y` are equal or not.

10.2.6 An Example Method

Let us examine a simple method definition in Scala. We wish to define a procedure to take in a number, add 10 to it and return the result.

```
object MethodTest {  
  
  def addTen(aNumber: Int): Int = {  
    var result = 0  
    result = aNumber + 10  
    return result  
  }  
  
}
```

Although the format may be slightly different from code that you have been used to, it is relatively straightforward. If you have C or C++ experience you might think that it is exactly the same as what you have seen before. Be careful with that idea—things are not always what they seem!

Let us look at some of the constituent parts of the method definition. The method name is `addTen`. In this case, the method has one parameter, called `aNumber`, of the basic type `Int`. Just, and as in any other language, the parameter variable is limited to the scope of this method (and is a `val`). The method also defines a temporary variable, `result`, also of the basic type `Int` and limited to the scope of this method (and this is a `var`).

Variable names are identifiers that contain only letters and numbers and must start with a letter (the underscore, `_`, and the dollar sign, `$`, count as letters). Some examples are:

```
anObject MyCar totalNumber $total
```

A capitalisation convention is used consistently throughout Java and most Scala programmers adhere to this standard:

- *Private variables and methods* (i.e. instance or temporary variables and almost all methods) start with a lower case letter.
- *Shared constants* are all in upper case.
- *Class* always start with an upper case letter.

Another convention is that if a variable or method name combines two or more words, then you should capitalise the first letter of each word, from the second word

onwards, e.g. `displayTotalPay`, `returnStudentName`. This is referred to as modified Camel Case.

10.2.7 Overriding `toString`

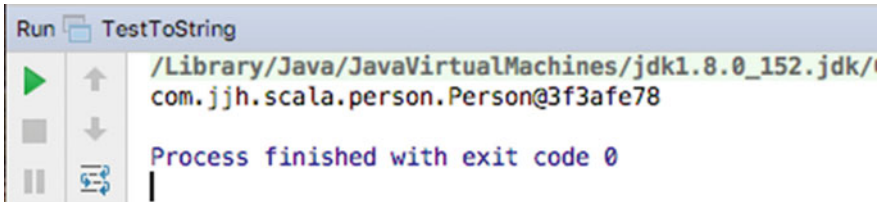
One of the facilities that is available for all types is the ability to convert itself to a string. This is particularly useful when printing an instance out (i.e. to help with debugging scenarios). The `println` functionality we have been using is written in such a way that if it is given an instance to print, it will ask that instance to convert itself to a string and then print that string. It does this by calling a method called `toString` on the instance. Given the following class, we can therefore print the string representation of instances to the console:

```
class Person(val name: String, var age: Int)
```

This can be shown by the following test harness application:

```
object TestToString extends App {
  val p = new Person("John", 49)
  println(p)
}
```

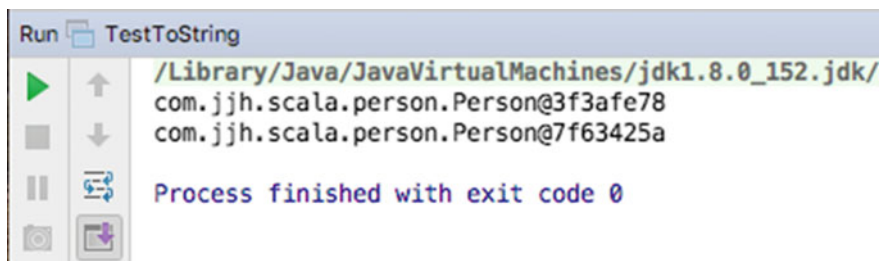
The result of running this program is shown in the console of the Eclipse IDE. However, the output might not be what you expect. The following diagram presents an example of the default output generated by `toString`.



As you can see from this example, the *default* behaviour for an object is to convert itself into a string version based on the fully qualified class name (i.e. `com.jjh.scala.person.Person`), followed by an '@' sign, followed by the hashcode for the object (the hexadecimal number following the '@'). The hashcode should be unique and allows us to distinguish between one instance of a class and another, for example,

```
object TestToString extends App {
  val p1 = new Person("John", 49)
  println(p1)
  val p2 = new Person("Denise", 46)
  println(p2)
}
```

The result of executing this application is shown in the next figure.



As you can see, the hexadecimal numbers following the ‘@’ are different. However, this is not very useful when need to distinguish between the instance representing *John* and the instance representing *Denise*.

The problem is that the default `toString` behaviour is defined at a more *abstract* level than the class `Person`. That is, the default behaviour does not know about the name and age properties. We can overcome this problem by redefining the way in which instances of the class `Person` convert themselves to a `String`. We do this by redefining the `toString` method mentioned earlier.

For example, in the following listing, we have redefined the `toString` to return a string constructed from the string “Person”, followed by the instances current values for name and age:

```
class Person(val name: String, var age: Int) {
  override def toString() = s"Person: $name: $age"
}
```

Be very careful how you define this method. It must be called `toString` (with a capital ‘S’). Scala is very case sensitive, and the method `tostring` and the method `toString` are two completely different methods. As we are redefining the default behaviour for `toString`, we must make sure the spelling and capitalisation are the same. Also note that we must use the keyword `override` before the `def` keyword to indicate we are expecting to be redefining the default method (it is called `override` as it is actually via inheritance that we obtain the default implementation of `toString`, but we will return to this in the chapter that focuses on inheritance). Also, note that the `toString` method must return a `String`!

Now when we rerun our simple application the output is modified such that we now obtain a far more meaningful result:

```

Run TestToString
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/
Person: John: 49
Person: Denise: 46
Process finished with exit code 0

```

10.2.8 Defining Property Methods

In the previous section we described `name` and `age` as properties of the instance `p1`, but what does this mean? A Property is an item of data, held within an instance of a class, that can be accessed externally to that instance either as a read-only property or as a read/write property.

Essentially Scala creates a reader (also known as a getter) method and a writer (also known as a setter) method associated with each property. If the properties were marked as *vals*, then it would only create the reader methods.

Depending upon the context in which you reference the property, Scala knows whether to invoke the reader or writer. For example if you are attempting to access the value of the property then it knows to invoke the reader, whereas if you are attempting to set the value of the property it knows to use the writer method.

Scala also allows a programmer to override the default readers and writers if required; it is just that the default behaviour provided by Scala generally meets the requirements of most developers.

If you wish to define your own readers and writers (or to help understand what is being created for you) then there are a few additional things to understand. The first is that the properties you have defined are by default public—that is visible outside of your instances to anything within the Scala world. An alternative would have been to mark them as private (note you do not need to say anything for them to be made public, but you need to make a conscientious decision to make them private).

The second thing you need to be aware of is that Scala does not actually distinguish between a property, a method or a function to any great extent, and it is the way that it is defined and invoked which actually allows Scala to work out what you want. Therefore if you are defining your own readers and writers then you will need to ensure that the name of the field that will hold the data is different to the name of the methods used to access that field. Although any name could be used, by convention the field name is prefixed by an underbar (`'_'`).

Thirdly to distinguish between a method that should be used on the left-hand side of an assignment and one that should be used to retrieve a value, a writer method is post fixed by an underbar ('_').

Given the above, `Person2` is a class that defines the same behaviour as `Person1` but we have done it longhand ourselves rather than rely on Scala to create the getters and setter methods for us:

```
class Person2 {  
  
    private var _name = ""  
    private var _age = 0  
  
    // Getters  
    def age = _age  
    def name = _name  
  
    // Setters  
    def age_=(value: Int): Unit = _age = value  
    def name_=(value: String): Unit = _name = value  
  
}
```

In the above code we have created two private properties `_name` and `_age`. These are accessed by two methods each; one to return the value (the getter) and one to set the value (the setter).

For example, the `def age` method returns the value of `_age`. Note it could have been written longhand as:

```
def age(): Int = {  
    return _age  
}
```

However we are using Scala's ability to infer much of the template from above and thus merely need to write:

```
def age = _age
```

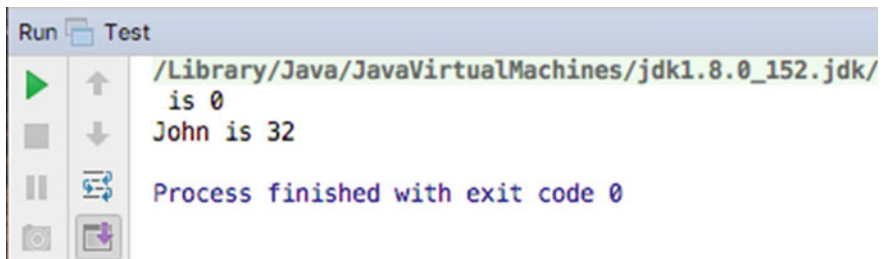
The setter methods are a little more complex. We have had to call the methods `age_` and `name_`. They are defined to take a value (of an appropriate type which we are explicitly specifying here). And we are indicating that they do not themselves return anything (hence the `Unit` return type). Within the body of the methods we then indicate that the value passed in is assigned to the appropriate property. Even so this is still a shorthand for the longhand from which would be (for the age setter method):

```
def age_(value: Int): Unit = {
  _age = value
}
```

We can now use the same test program with this class as we have previously used for the `Person1` class:

```
object Test extends App {
  val p = new Person2()
  println(s"${p.name} is ${p.age}")
  p.name = "John"
  p.age = 32
  println(s"${p.name} is ${p.age}")
}
```

And it produces the output shown below.



```
Run Test
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/
is 0
John is 32
Process finished with exit code 0
```

This approach may be useful if you wish to add some non-default behaviour to either the setter or getter methods.

10.3 Named Parameters

In most situations, when you invoke a constructor, a method or a function each argument is matched, in sequence with the parameters of the constructor, method or function. Thus given the method `mult` in the object `Processor`:

```
object Processor {
  def mult(x: Int, y: Int): Int = x * y
}
```

Then we can invoke this method as follows:

```
object SampleApp extends App {
  val r1 = Processor.mult(2, 3)
  println(r1)
}
```

In this case the value 2 is bound to the parameter 'x' and the value 3 is bound to the parameter `y` and thus we multiply 2 by 3 to obtain 6.

However, an alternative approach is to use the names of the parameters. Named parameters allow you to pass in argument to a constructor, method or function as name–value pairs. These pairs can be in any order, and Scala will work out how to bind them. The syntax for this is based on `name=value`, with each parameter separated by a comma (`,`). For example, the above invocation of `mult` could be rewritten as:

```
object SampleApp extends App {
  val r2 = Processor.mult(x=2, y=3)
  println(r2)
}
```

Now we are explicitly binding the value 2 to `x` and the value 3 to `y`. The end result is that the value 6 is again printed out. However, as the order is no longer significant we could also write:

```
object SampleApp extends App {
  val r3 = Processor.mult(y=3, x=2)
  println(r3)
}
```

This again binds the value 2 to `x` and the value 3 to `y` and once again results in the value 6 being printed out.

Thus the order of the parameters is no longer significant. Note that you can also mix positional arguments with named arguments (in which case the positional arguments come first), for example,

```
object SampleApp extends App {
  val r4 = Processor.mult(2, y=3)
  println(r4)
}
```

Named parameters are most often used with default parameters. This allows the optional values to be used for all omitted parameters, but the named parameters to be used for those to be specified. For example,

```
import java.util.Date

class Activity(val date: Date = new Date(),
              var title: String = "activity",
              var owner: String = "anon",
              var live: Boolean = true) {
  override def toString(): String = {
    "Activity[" +
      date + ", " +
      title + ", " +
      owner + ", " +
      live + "]"
  }
}
```

The class `Activity` defines a primary constructor that takes four parameters. Each of these parameters has a default value. However, if we used position-based parameters then we could not just provide the second, third or fourth parameter when we create an `Activity`. However, using named parameters allows us to do exactly this. For example, to create a new `Activity` with the owner set to “John” but with the defaults used for the other three parameters we can write:

```
object SampleApp extends App {
  val a1 = new Activity(owner = "John")
  println(a1)
}
```

The result of running this code is shown below:

```
Activity[Wed Dec 13 14:29:50 GMT 2017, activity, John,
true]
```

As you can see the default values for `date`, `title` and `live` have been used with the owner set to “John”.

It is also common to find that the use of named parameters is used with the alternative curly bracket ‘{ }’ syntax used for parentheses. This form results in a construct that looks more as if it is part of the language than a user-defined type. For example, using the alternative syntax we can create a new `Activity` specifying the type of activity and the owner (as “Presentation” and “Denise”, respectively).

```
val a2 = new Activity {
  owner = "Denise"
  title = "Presentation"
}
println(a2)
```

The result of running this is shown below:

```
Activity[Wed Dec 13 14:31:20 GMT 2017, Presentation,
Denise, true]
```

Note that the order of `owner` and `title` is not significant and that `date` and `live` are still defaulted. Also note that `Activity` now appears to be a language construct.