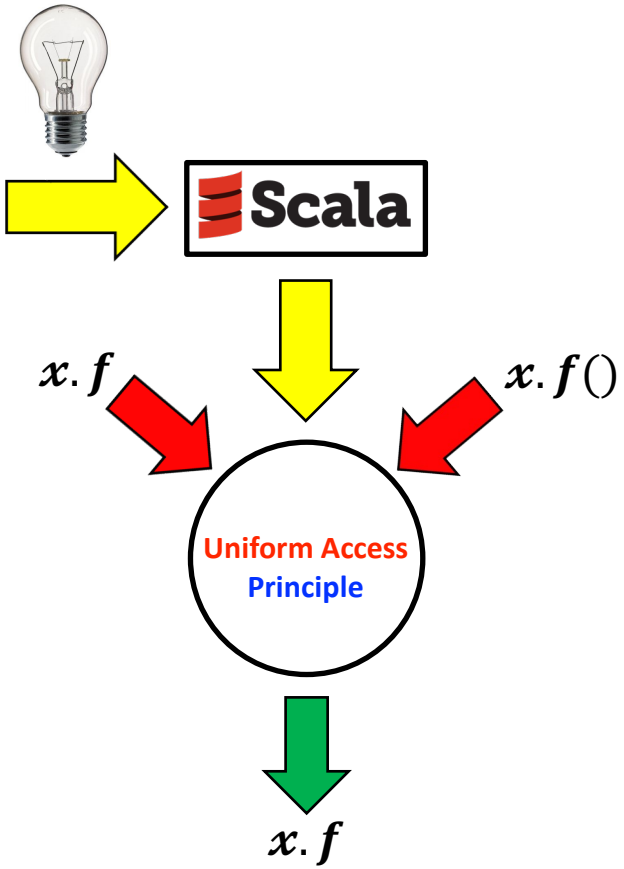
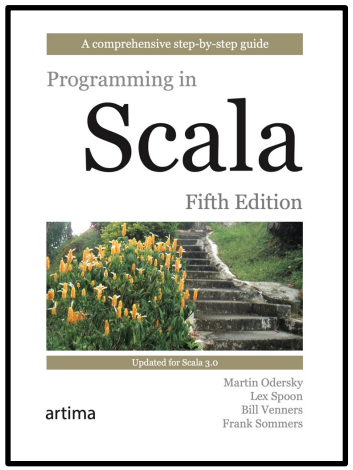
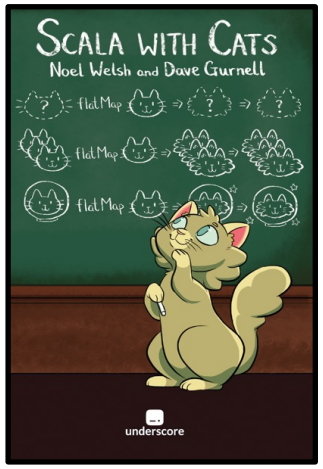
The Uniform Access Principle







Bertrand Meyer
@Bertrand_Meyer

Martin Odersky
@odersky

Noel Welsh
@noelwelsh

Dave Pereira-Gurnell
@davegurnell

Daniel Westheide
@kaffeecoder

slides by  @philip_schwarz slideshare https://www.slideshare.net/pjschwarz

Let’s begin by looking at the definition and explanation of the **Uniform Access Principle** in **Bertrand Meyer**’s book **Object-Oriented Software Construction**.

@philip_schwarz

Uniform Access

Although it may at first appear just to address a notational issue, the Uniform Access principle is in fact a design rule which influences many aspects of object-oriented design and the supporting notation.

…

Let x be a name used to access a certain data item (what will later be called an object) and f the name of a feature applicable to x. (A feature is an operation; this terminology will also be defined more precisely.)

For example, x might be a variable representing a bank account, and f the feature that yields an account's current balance.

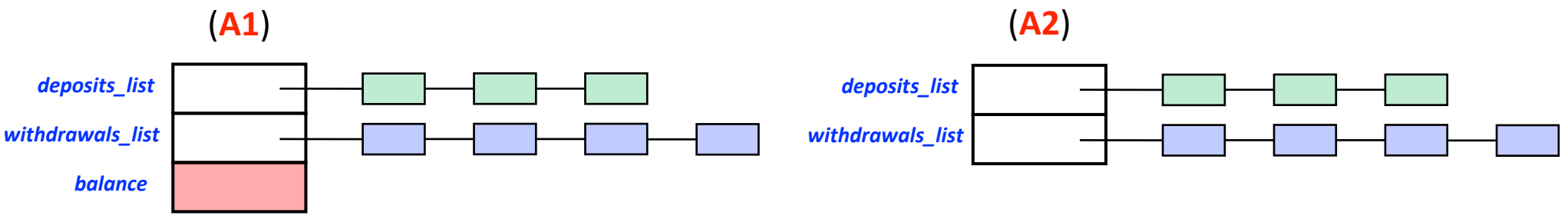
Uniform Access addresses the question of how to express the result of applying f to x, using a notation that does not make any premature commitment as to how f is implemented.

In most design and programming languages, the expression denoting the application of f to x depends on what implementation the original software developer has chosen for feature f: is the value stored along with x, or must it be computed whenever requested? Both techniques are possible in the example of accounts and their balances:

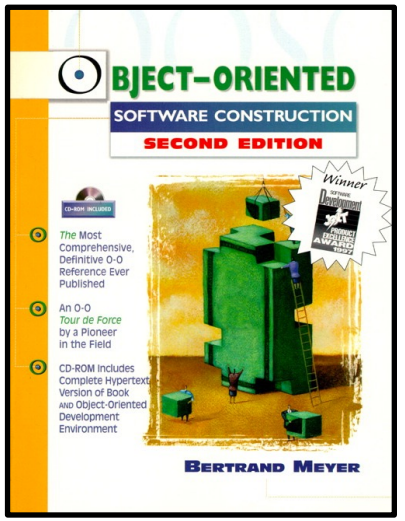
A1 • You may represent the balance as one of the fields of the record describing each account, as shown in the figure. With this technique, every operation that changes the balance must take care of updating the balance field.

A2 • Or you may define a function which computes the balance using other fields of the record, for example fields representing the lists of withdrawals and deposits. With this technique the balance of an account is not stored (there is no balance field) but computed on demand.

Two representations for a bank account



A common notation, in languages such as Pascal, Ada, C, C++ and Java, uses $x.f$ in case A1 and $f(x)$ in case A2.



Bertrand Meyer

@Bertrand_Meyer

Choosing between representations A1 and A2 is a space-time tradeoff: one economizes on computation, the other on storage.

The resolution of this tradeoff in favor of one of the solutions is typical of representation decisions that developers often reverse at least once during a project's lifetime.

So for continuity's sake it is desirable to have a feature access notation that does not distinguish between the two cases; then if you are in charge of x's implementation and change your mind at some stage, it will not be necessary to change the modules that use f.

This is an example of the Uniform Access principle.

In its general form the principle may be expressed as:

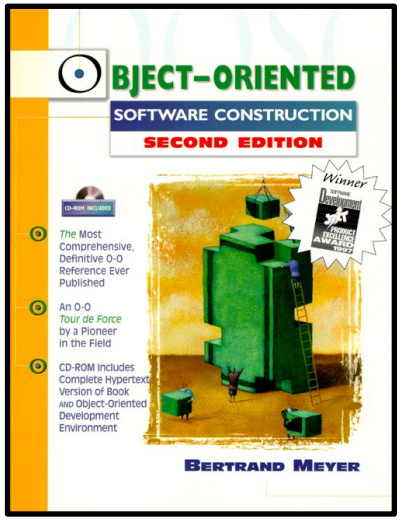
Uniform Access principle

All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation.

Few languages satisfy this principle.

An older one that did was Algol W, where both the function call and the access to a field were written $a(x)$.

Object-oriented languages should satisfy Uniform Access, as did the first of them, Simula 67, whose notation is $x.f$ in both cases.



Bertrand Meyer
@Bertrand_Meyer

One of the few languages that support the **Uniform Access Principle** is **Scala**.

10.3 Defining parameterless methods

As a next step, we'll add methods to **Element** that reveal its **width** and **height**, as shown in Listing 10.2.

```
abstract class Element:
  def contents: Vector[String]
  def height: Int = contents.length
  def width: Int = if height == 0 then 0 else contents(0).length
```

Listing 10.2 · Defining **parameterless methods** **width** and **height**.

The **height** method returns the number of lines in contents.

The **width** method returns the length of the first line, or if there are no lines in the element, returns zero. (This means you cannot define an element with a height of zero and a non-zero width.)

Note that none of Element's three methods has a parameter list, not even an empty one.

For example, instead of:

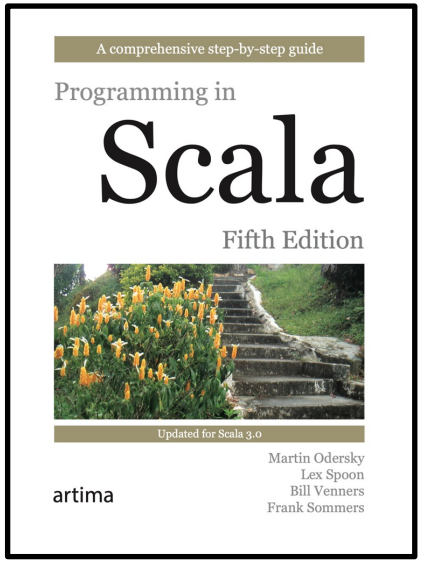
```
def width(): Int
```

the method is defined without parentheses:

```
def width: Int
```

Such *parameterless methods* are quite common in Scala.

By contrast, methods defined with empty parentheses, such as 'def height(): Int', are called *empty-paren methods*.



Martin Odersky

@odersky

The recommended convention is to use a parameterless method whenever there are no parameters *and* the method accesses state only by reading fields of the containing object (in particular, it does not change mutable state).

This convention supports the *uniform access principle*,[1] which says that client code should not be affected by a decision to implement an attribute as a field or method.

For instance, we could implement width and height as fields, instead of methods, simply by changing the def in each definition to a val:

```
abstract class Element:
  def contents: Vector[String]
  val height: Int = contents.length
  val width: Int = if height == 0 then 0 else contents(0).length
```

The two pairs of definitions are completely equivalent from a client's point of view.

The only difference is that field accesses might be slightly faster than method invocations because the field values are pre-computed when the class is initialized, instead of being computed on each method call.

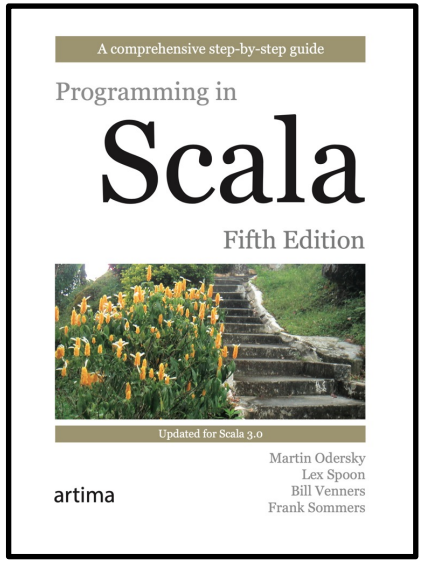
On the other hand, the fields require extra memory space in each Element object.

So it depends on the usage profile of a class whether an attribute is better represented as a field or method, and that usage profile might change over time.

The point is that clients of the Element class should not be affected when its internal implementation changes.

In particular, a client of class Element should not need to be rewritten if a field of that class gets changed into an access function, so long as the access function is *pure* (*i.e.*, it does not have any side effects and does not depend on mutable state).

[1] Meyer, *Object-Oriented Software Construction* [Mey00]



Bill Venners

@bvenners

The client should not need to care either way.

So far so good.

But there's still a slight complication with the way Java and Scala 2 handle things.

The problem is that Java does not implement the uniform access principle, and Scala 2 does not fully enforce it.

For example, it's `string.length()` in Java, not `string.length`, even though it's `array.length`, not `array.length()`.

This can be confusing.

To bridge that gap, Scala 3 is very liberal when it comes to mixing parameterless and empty-paren methods defined in Java or Scala 2.

In particular, you can override a parameterless method with an empty-paren method, and *vice versa*, so long as the parent class was written in Java or Scala 2.

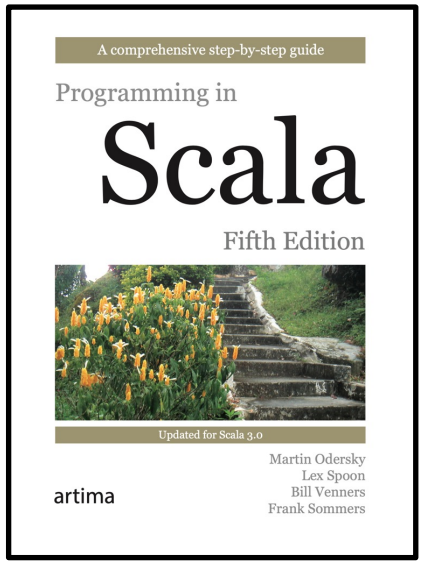
You can also leave off the empty parentheses on an invocation of any function defined in Java or Scala 2 that takes no arguments.

For instance, the following two lines are both legal in Scala 3:

```
Array(1, 2, 3).toString
"abc".length
```

In principle it's possible to leave out all empty parentheses in calls to functions defined in Java or Scala 2.

However, it's still recommended to write the empty parentheses when the invoked method represents more than a property of its receiver object.



Lex Spoon

For instance, empty parentheses are appropriate if the method performs I/O, writes reassignable variables (vars), or reads vars other than the receiver's fields, either directly or indirectly by using mutable objects.

That way, the parameter list acts as a visual clue that some interesting computation is triggered by the call.

For instance:

```
"hello".length // no () because no side-effect
println()      // better to not drop the ()
```

To summarize, it is encouraged in Scala to define methods that take no parameters and have no side effects as parameterless methods (*i.e.*, leaving off the empty parentheses).

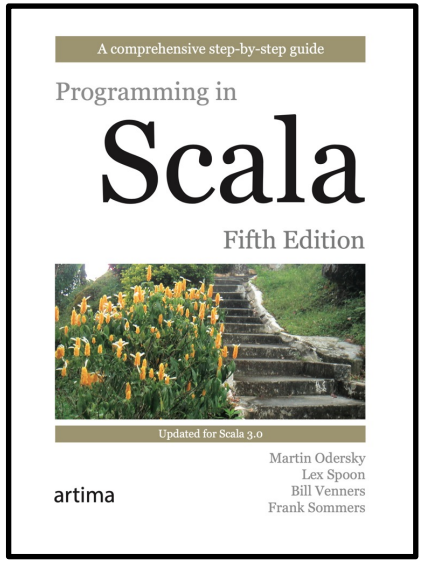
On the other hand, you should never define a method that has side-effects without parentheses, because invocations of that method would then look like a field selection. So your clients might be surprised to see the side effects.

Similarly, whenever you invoke a function that has side effects, be sure to include the empty parentheses when you write the invocation, even if the compiler doesn't force you.[2]

Another way to think about this is if the function you're calling performs an operation, use the parentheses.

But if it merely provides access to a property, leave the parentheses off.

[2] The compiler requires that you invoke parameterless methods defined in Scala 3 without empty parentheses and empty-parens methods defined in Scala 3 with empty parentheses.



Frank Sommers

In preparation for the rest of this deck, let's see how **Scala with Cats** describes the following terms for **models of evaluation**:

- **Eager**
- **Lazy**
- **Memoized**

4.6.1 Eager, Lazy, Memoized, Oh My!

What do these terms for models of evaluation mean?

Let's see some examples.

Let's first look at Scala vals.

We can see the evaluation model using a computation with a visible side-effect.

In the following example, the code to compute the value of x is executed at the place where it is defined rather than on access.

Accessing x recalls the stored value without re-running the code.

```
val x = {
  println("Computing X")
  math.random
}
// Computing X
// x: Double = 0.15241729989551633

x // first access
// res0: Double = 0.15241729989551633 // first access
x // second access
// res1: Double = 0.15241729989551633
```

This is an example of call-by-value evaluation:

- the computation is evaluated at the point where it is defined (eager); and
- the computation is evaluated once (memoized).

Noel Welsh
@noelwelsh

Dave Pereira-Gurnell
@davegurnell

Let's look at an example using a def.

The code to compute y below is not run until we use it, and is re-run on every access:

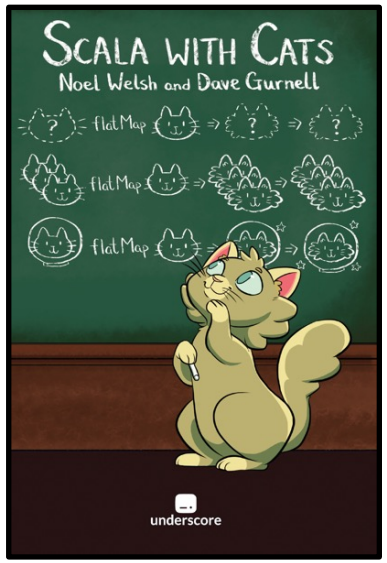
```
def y = {
  println("Computing Y")
  math.random
}

y // first access
// Computing Y
// res2: Double = 0.5270290953284378 // first access
y // second access
// Computing Y
// res3: Double = 0.348549829974959
```

These are the properties of call-by-name evaluation:

- the computation is evaluated at the point of use (lazy); and
- the computation is evaluated each time it is used (not memoized).

Noel Welsh
@noelwelsh

Dave Pereira-Gurnell
@davegurnell



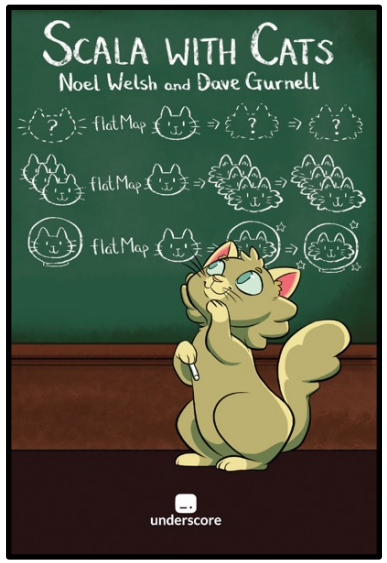
Last but not least, lazy vals are an example of call-by-need evaluation.

The code to compute z below is not run until we use it for the first time (lazy). The result is then cached and re-used on subsequent accesses (memoized):

```
lazy val z = {
  println("Computing Z")
  math.random
}

z // first access
// Computing Z
// res4: Double = 0.6672110951657263 // first access
z // second access
// res5: Double = 0.6672110951657263
```

Noel Welsh
@noelwelsh

Dave Pereira-Gurnell
@davegurnell



Let's summarize.

There are two properties of interest:

- evaluation at the point of definition (eager) versus at the point of use (lazy); and
- values are saved once evaluated (memoized) or not (not memoized).

There are three possible combinations of these properties:

- call-by-value which is eager and memoized;
- call-by-name which is lazy and not memoized;and
- call-by-need which is lazy and memoized.

The final combination, eager and not memoized, is not possible.

Noel Welsh
@noelwelsh

Dave Pereira-Gurnell
@davegurnell

Here is another summary of the concepts that we have just gone through.

call-by-value

```
scala> val x = {
     |   println("Computing X")
     |   math.random
     | }
Computing X
val x: Double = 0.5835699271495728

scala> x
val res56: Double = 0.5835699271495728

scala> x
val res57: Double = 0.5835699271495728
```

eager and memoized

call-by-name

```
scala> def y = {
     |   println("Computing Y")
     |   math.random
     | }
def y: Double

scala> y
Computing Y
val res58: Double = 0.6406566851969714

scala> y
Computing Y
val res59: Double = 0.13912093420520477
```

lazy and not memoized

call-by-need

```
scala> lazy val z = {
     |   println("Computing Z")
     |   math.random
     | }
lazy val z: Double

scala> z
Computing Z
val res60: Double = 0.059971734095200735

scala> z
val res61: Double = 0.059971734095200735
```

lazy and memoized

@philip_schwarz

Armed with that understanding of the terms **Eager**, **Lazy** and **Memoized**, let's see how **Scala from Scratch** factors **laziness** and **memoization** into the **uniform access principle**.

Overriding inherited methods or fields

Inheriting from a super class means that you have access to all its methods and fields, as long as they are not declared as private (we will discuss visibility modifiers a bit later in this chapter). It also means that you can override them.

To demonstrate this, let's override the `toString` method defined in `AnyRef`, or rather `java.lang.Object`. To override a method defined in a parent class, you need to prefix it with the override modifier, like so:

```
class Position(val x: Int, val y: Int) {
  // rest of the class body omitted
  override def toString: String =
    "%s (x: %d, y: %d)".format(super.toString, x, y)
}
```

Here you can see how to access a field or method defined in the parent class. Just as in most object-oriented languages, you have to use the super keyword to reference the parent.

In this example, we do this in order to make use of the already existing `toString` implementation in `AnyRef`.

With our custom `toString` implementation in place, when we create a `Position` instance in the Scala REPL it's now shown as something like this:

```
scala> val pos = new Position(3, 2)
val pos: Position = Position@28a2283d (x: 3, y: 2)
```

That is because the REPL calls `toString` on anything it needs to display.

We can also call the `toString` method directly:

```
scala> val s = pos.toString
val s: String = Position@385d7101 (x: 3, y: 2)
```

Daniel Westheide
@kaffeecoder

The uniform access principle

Calling the `toString` method looks exactly the same as if our class had a field called `toString`. Instead of overriding `toString` as we did before, we can also do it like this:

```
class Position(val x: Int, val y: Int) {
  // rest of the class body omitted
  override val toString: String =
    "%s (x: %d, y: %d)".format(super.toString, x, y)
}
```

From the outside, this looks the same. To verify, please start a new REPL session after adjusting your `Position` class and try again:

```
scala> val pos = new Position(3, 2)
val pos: Position = Position@28a2283d (x: 3, y: 2)

scala> val s = pos.toString
val s: String = Position@385d7101 (x: 3, y: 2)
```

This is what we call the *uniform access principle*.

Parameterless methods and values defined in a class are accessed in a uniform way, and you can consider both of them to be fields, or properties.

The reason why this can work is that value definitions and method definitions in a class live in the same namespace.

Whether you want to use a val or a def, a value or a method, largely depends on how you expect that field to be used, and how expensive it is to compute the result.

Daniel Westheide
@kaffeecoder

Implementing `toString` as a value means that the string formatting necessary to create the result of `toString` will happen immediately when an instance of the Position class is created.

It will happen regardless of whether `toString` is ever used by anyone, on that Position instance.

Implementing it as a def means that all this string concatenation will only happen if and when someone actually accesses `toString`.

In this case, the resulting string will be recomputed every time someone accesses `toString`, even though the class is immutable — which means that the result of `toString` will always be the same for one instance of Position.

As always, it's a tradeoff.

In a real program, you would usually not use a val to implement `toString`, but it is often a reasonable choice for other methods.

Daniel Westheide
@kaffeecoder

Uniform access principle revisited

In Section 3.1, you learned about the uniform access principle.

Methods and values share the same namespace, so it doesn't make a difference for a user of a class whether a field of that class is defined as a value or as a parameterless method.

Now that you have learned about lazy values, you should be aware that it also doesn't matter whether a field is a strict value or a lazy value — the uniform access principle means that all three cases are the same from a consumer's point of view.

Going back to our boardgame sbt project from Chapter 3, you may choose to override the `toString` method not as a regular, strict val, but as a lazy val, like this:

```
class Position(val x: Int, val y: Int) {
  // rest of the class body omitted
  override lazy val toString: String =
    "%s [x: %d, y: %d]".format(super.toString, x, y)
}
```

This can make sense if the computation of the String is quite expensive, and it's unclear whether anyone will never call `toString` on instances of the respective class.

The same is true for any other fields in classes you define: You can be flexible about whether to use a val, lazy val, or a def. Which of the three makes most sense depends a lot on your specific use case.

The other side of the coin is that, just as with by-name parameters, when looking at some code accessing a field, there is no way of knowing the evaluation strategy for that field without navigating to the source code in which the field is defined.

Daniel Westheide
@kaffeecoder

Here is a recap

parameterless method

```
class Position(val x: Int, val y: Int) {
  // rest of the class body omitted
  override def toString: String =
    "%s (x: %d, y: %d)".format(super.toString, x, y)
}
```

lazy and not memoized

value

```
class Position(val x: Int, val y: Int) {
  // rest of the class body omitted
  override val toString: String =
    "%s (x: %d, y: %d)".format(super.toString, x, y)
}
```

eager and memoized

lazy value

```
class Position(val x: Int, val y: Int) {
  // rest of the class body omitted
  override lazy val toString: String =
    "%s [x: %d, y: %d]".format(super.toString, x, y)
}
```

lazy and memoized

uniform access

```
scala> val pos = new Position(3, 2)
val pos: Position = Position@28a2283d (x: 3, y: 2)

scala> val s = pos.toString
val s: String = Position@385d7101 (x: 3, y: 2)
```

This is what we call the *uniform access principle*

Parameterless methods, values and lazy values defined in a class are accessed in a uniform way, and you can consider all of them to be fields, or properties.

You can be flexible about whether to use a val, lazy val, or a def.

Which of the three makes most sense depends a lot on your specific use case.

Daniel Westheide
@kaffeecoder

That’s all. I hope you found it useful.

@philip_schwarz