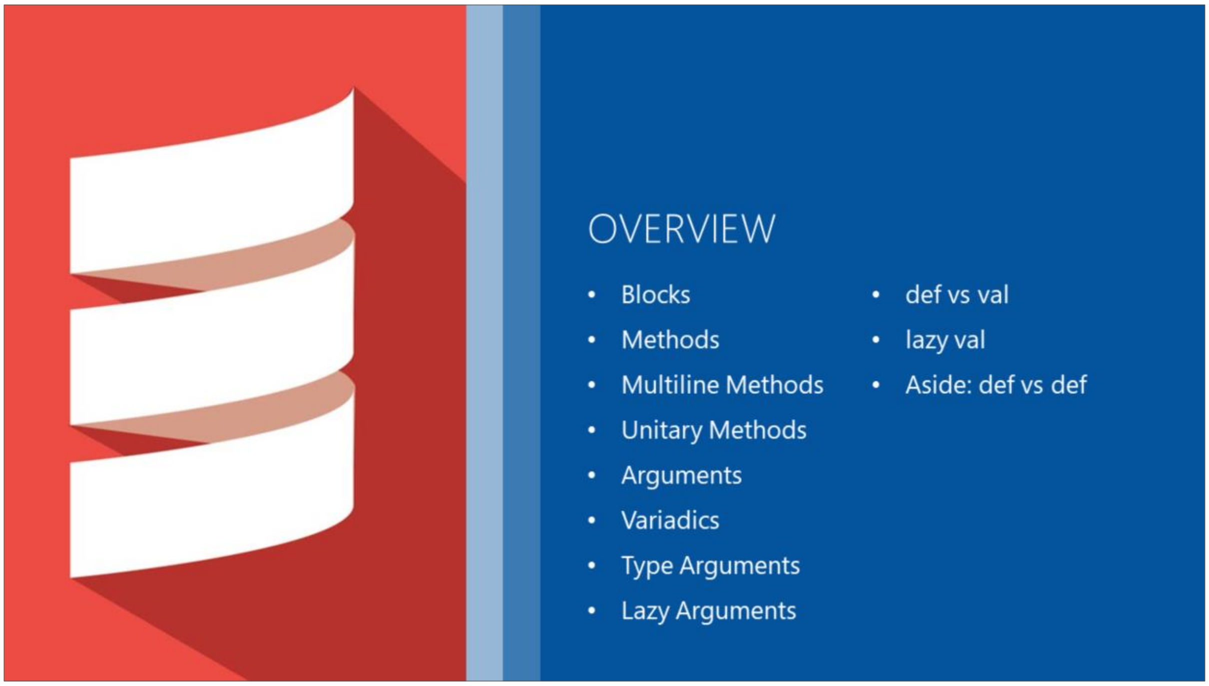
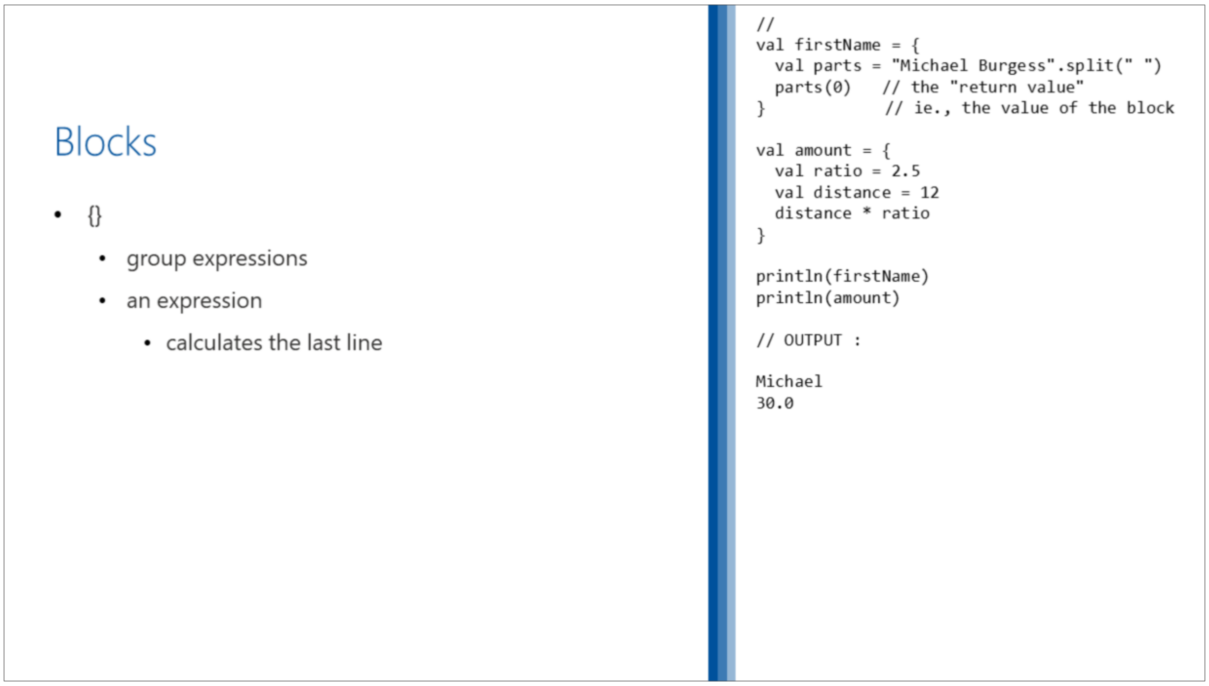
Methods

SCALA PROGRAMMING
A DATA SCIENCE AND MACHINE LEARNING COURSE



OVERVIEW

- Blocks
- Methods
- Multiline Methods
- Unitary Methods
- Arguments
- Variadics
- Type Arguments
- Lazy Arguments
- def vs val
- lazy val
- Aside: def vs def



Blocks

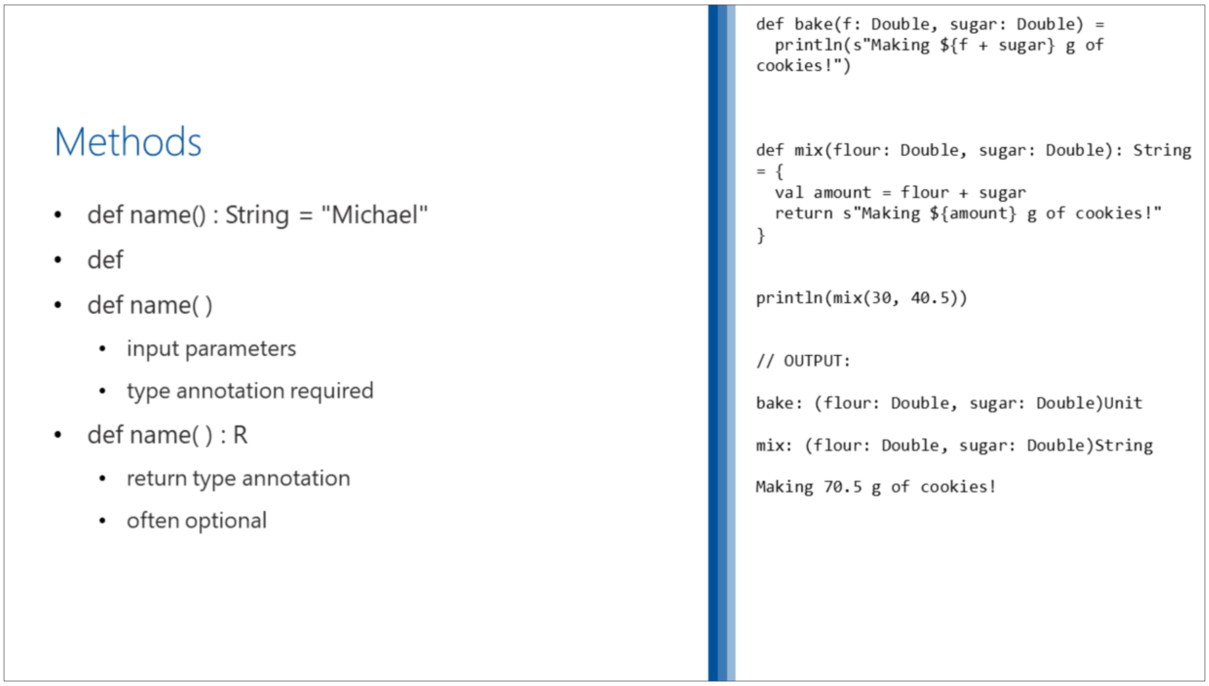
- {}
  - group expressions
  - an expression
    - calculates the last line

```
//
val firstName = {
  val parts = "Michael Burgess".split(" ")
  parts(0)   // the "return value"
}            // ie., the value of the block

val amount = {
  val ratio = 2.5
  val distance = 12
  distance * ratio
}

println(firstName)
println(amount)

// OUTPUT :

Michael
30.0
```

Braces group expressions into blocks

A block is an expression

The last line of the block is the value of the entire block



Methods

- def name() : String = "Michael"
- def
- def name( )
  - input parameters
  - type annotation required
- def name( ) : R
  - return type annotation
  - often optional

```
def bake(f: Double, sugar: Double) =
  println(s"Making ${f + sugar} g of
cookies!")


def mix(flour: Double, sugar: Double): String
= {
  val amount = flour + sugar
  return s"Making ${amount} g of cookies!"
}


println(mix(30, 40.5))


// OUTPUT:

bake: (flour: Double, sugar: Double)Unit

mix: (flour: Double, sugar: Double)String

Making 70.5 g of cookies!
```

Methods are named blocks of code

  a label following def

  accepting a set of input parameters

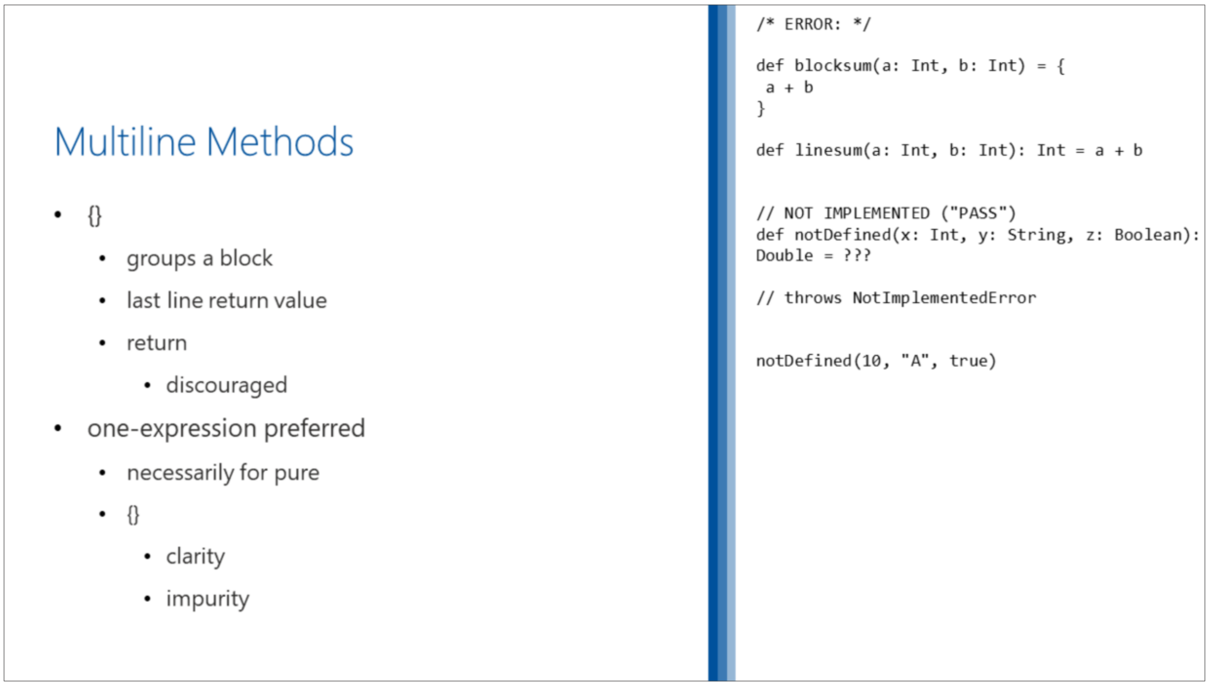
  each must have its type after the type-marker :

  the return type is option

    precedes the = sign if present

The type signature of the method

  its parameter list followed by its return value



Multiline Methods

- {}
  - groups a block
  - last line return value
  - return
    - discouraged
- one-expression preferred
  - necessarily for pure
  - {}
    - clarity
    - impurity

```
/* ERROR: */

def blocksum(a: Int, b: Int) = {
 a + b
}

def linesum(a: Int, b: Int): Int = a + b


// NOT IMPLEMENTED ("PASS")
def notDefined(x: Int, y: String, z: Boolean):
Double = ???

// throws NotImplementedError


notDefined(10, "A", true)
```

Methods are assignments to blocks of code

a block may be one or more lines

multiple lines are grouped by braces

the last line is the value of the entire block

ie., the value returned

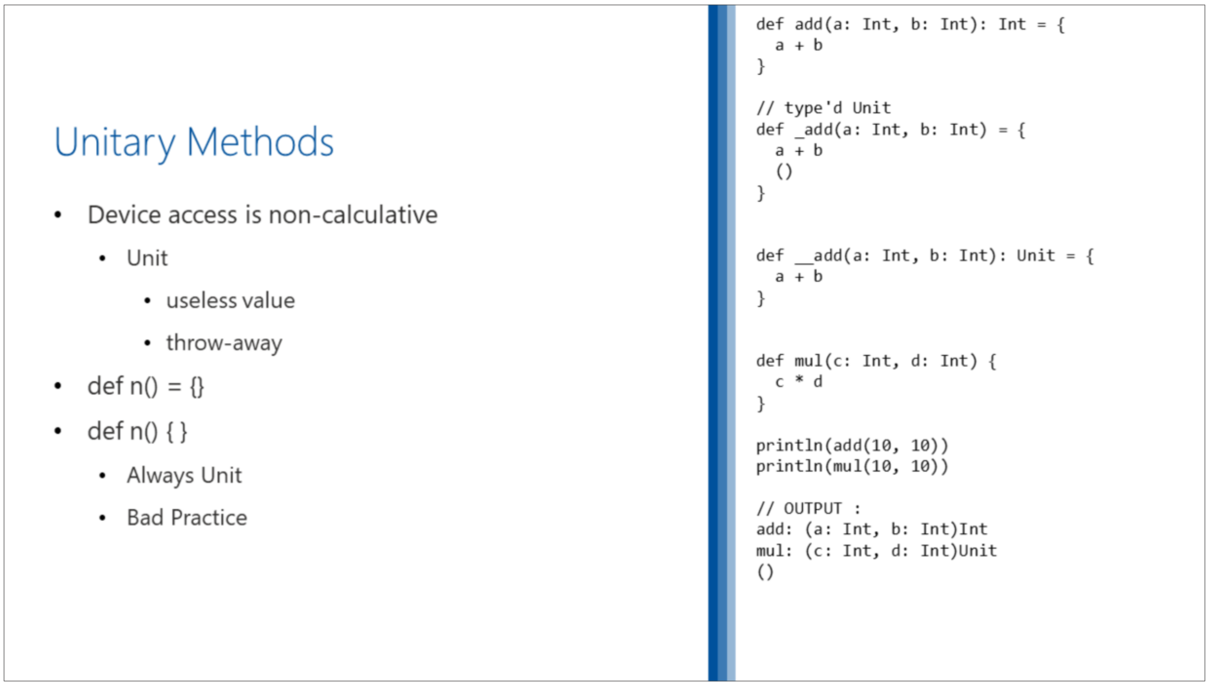
the return keyword may be used by it is strongly discouraged

Methods should be one-expression where possible

all pure functions are single expressions

multiple may be used to clarify

or if it needs to be impure



Unitary Methods

- Device access is non-calculative
  - Unit
    - useless value
    - throw-away
- def n() = {}
- def n() { }
  - Always Unit
  - Bad Practice

```
def add(a: Int, b: Int): Int = {
  a + b
}

// type'd Unit
def _add(a: Int, b: Int) = {
  a + b
  ()
}


def __add(a: Int, b: Int): Unit = {
  a + b
}


def mul(c: Int, d: Int) {
  c * d
}

println(add(10, 10))
println(mul(10, 10))

// OUTPUT :
add: (a: Int, b: Int)Int
mul: (c: Int, d: Int)Unit
()
```

Methods may not have meaningful value to return

in this case return the Unit value ()

or type the method : Unit

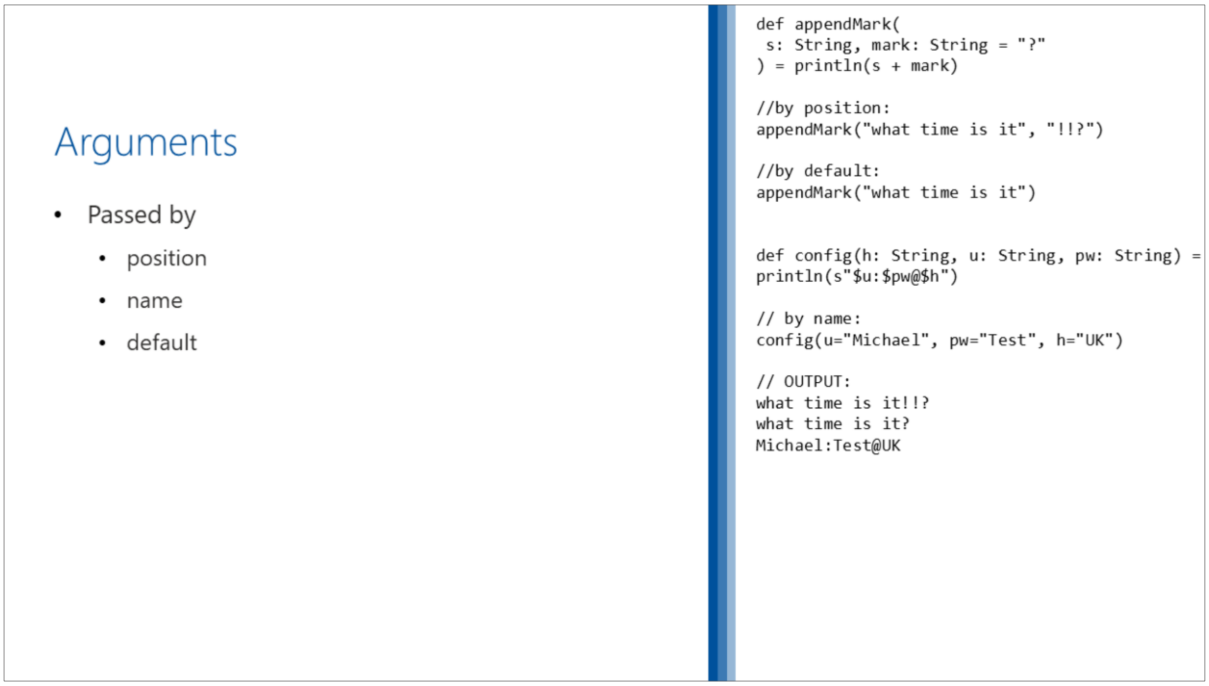
Unit-typed methods throw away their return value

Methods may be declared without the = symbol

these always return Unit

this is very bad practice as its unclear and subtle

will be removed in future versions



Arguments

- Passed by
  - position
  - name
  - default

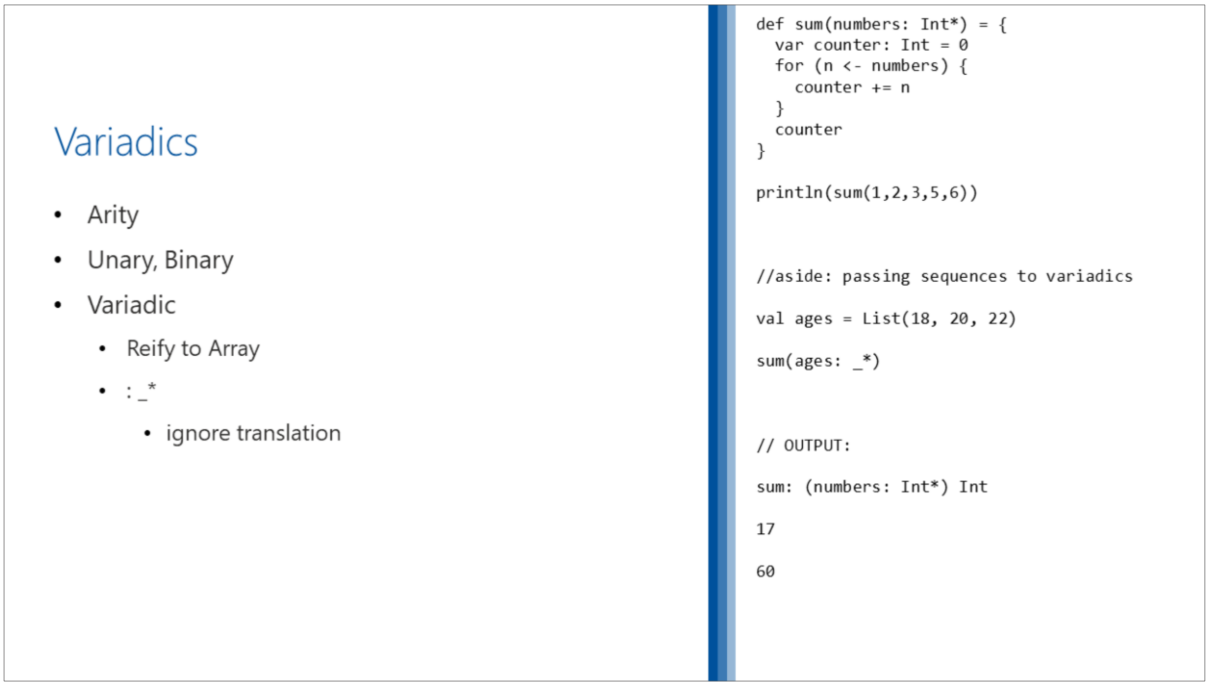
```
def appendMark(
 s: String, mark: String = "?"
) = println(s + mark)

//by position:
appendMark("what time is it", "!!?")

//by default:
appendMark("what time is it")


def config(h: String, u: String, pw: String) =
println(s"$u:$pw@$h")

// by name:
config(u="Michael", pw="Test", h="UK")

// OUTPUT:
what time is it!!?
what time is it?
Michael:Test@UK
```

Arguments may be passed by

position: arguments are passed in the order the method defines

name: passed using their name, in any order

default: if no argument is passed a default may be used



Variadics

- Arity
- Unary, Binary
- Variadic
  - Reify to Array
  - : _*
    - ignore translation

```
def sum(numbers: Int*) = {
  var counter: Int = 0
  for (n <- numbers) {
    counter += n
  }
  counter
}

println(sum(1,2,3,5,6))


//aside: passing sequences to variadics

val ages = List(18, 20, 22)

sum(ages: _*)


// OUTPUT:

sum: (numbers: Int*) Int

17

60
```

A method may accept a variable number of parameters

the arity of a function is how many parameters it has

a vari-adic function accepts a variable number

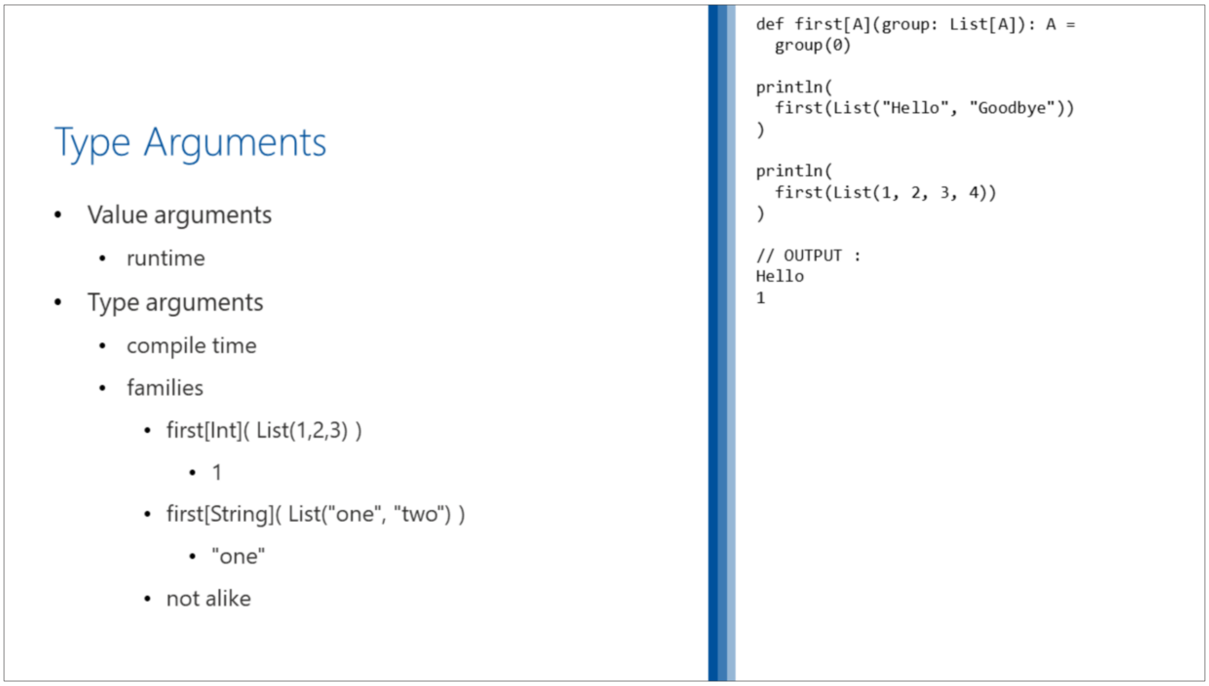
The compiler will actually pass any array as a single parameter

the parameter must have a type which ends with *

Sequences cannot be passed to varidic methods as-if they accepted one parameter

the compiler would wrap them in an array again

type the sequence _* to expand as parameters


Type Arguments

- Value arguments
  - runtime
- Type arguments
  - compile time
  - families
    - first[Int](List(1,2,3))
      - 1
    - first[String](List("one", "two"))
      - "one"
    - not alike

```
def first[A](group: List[A]): A =
  group(0)

println(
  first(List("Hello", "Goodbye"))
)

println(
  first(List(1, 2, 3, 4))
)

// OUTPUT :
Hello
1
```

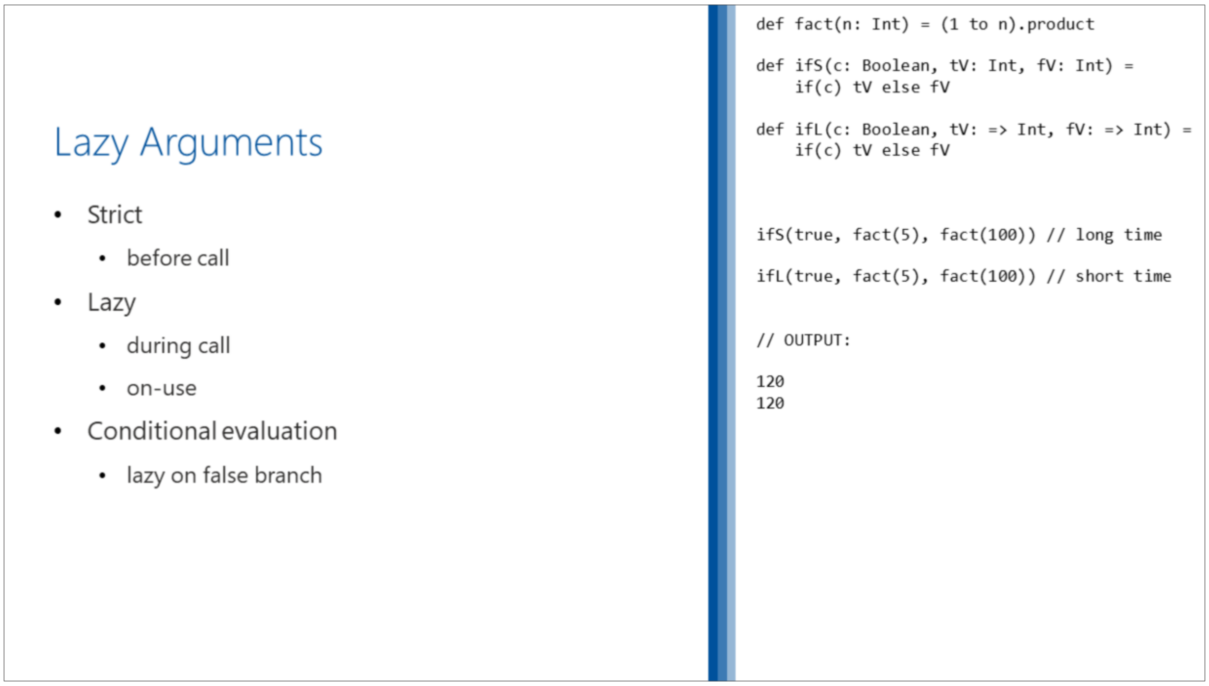
Methods may accept value arguments

object passed at runtime

Methods may accept type arguments

type information passed at compile time

Type information is used by the compiler to generate the correct runtime behaviour



Lazy Arguments

- Strict
  - before call
- Lazy
  - during call
  - on-use
- Conditional evaluation
  - lazy on false branch

```
def fact(n: Int) = (1 to n).product

def ifS(c: Boolean, tV: Int, fV: Int) =
    if(c) tV else fV

def ifL(c: Boolean, tV: => Int, fV: => Int) =
    if(c) tV else fV


ifS(true, fact(5), fact(100)) // long time

ifL(true, fact(5), fact(100)) // short time


// OUTPUT:

120
120
```

Arguments may be passed

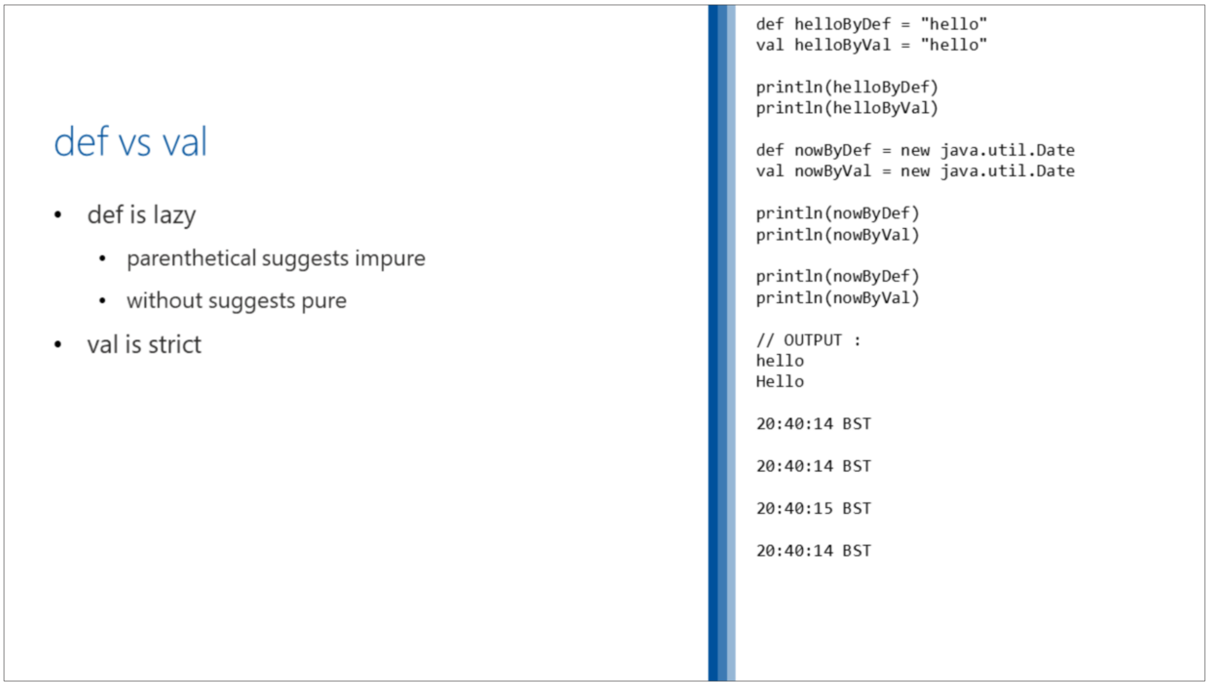
strictly: the usual behaviour where the compiler evaluates each argument before passing it to the method

lazily: the compiler delays evaluation of the arguments until they are first-read in the body of the function

Arguments passed lazily may never be evaluated

eg., if the argument is used on a false-branch it will never be read

this can make the execution of methods much more efficient


def vs val

- def is lazy
  - parenthetical suggests impure
  - without suggests pure
- val is strict

```
def helloByDef = "hello"
val helloByVal = "hello"

println(helloByDef)
println(helloByVal)

def nowByDef = new java.util.Date
val nowByVal = new java.util.Date

println(nowByDef)
println(nowByVal)

println(nowByDef)
println(nowByVal)

// OUTPUT :
hello
Hello

20:40:14 BST

20:40:14 BST

20:40:15 BST

20:40:14 BST
```

Declaring a variable with def makes it lazy

delaying the execution of the RHS until the variable is used

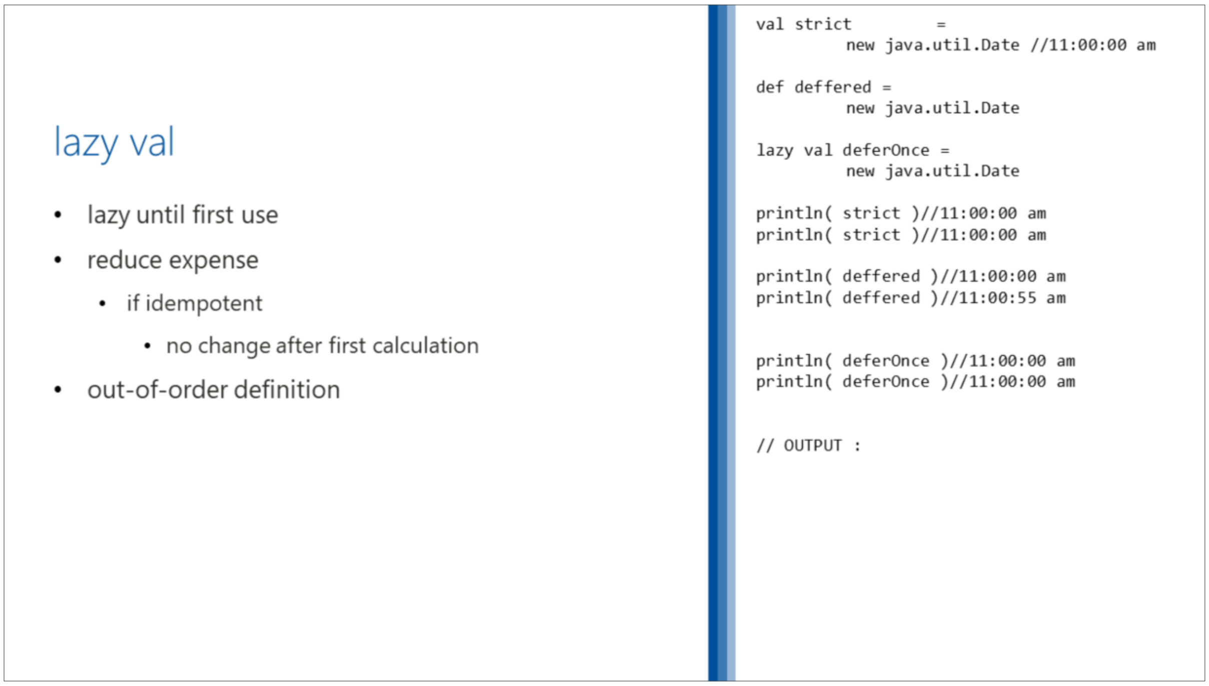
it is call-by-name

its value is produced when reading the variable

Declaring with val is strict

the RHS is evaluated on assignment

and never again



lazy val

- lazy until first use
- reduce expense
  - if idempotent
    - no change after first calculation
- out-of-order definition

```
val strict        =
        new java.util.Date //11:00:00 am

def deffered =
        new java.util.Date

lazy val deferOnce =
        new java.util.Date

println( strict )//11:00:00 am
println( strict )//11:00:00 am

println( deffered )//11:00:00 am
println( deffered )//11:00:55 am


println( deferOnce )//11:00:00 am
println( deferOnce )//11:00:00 am


// OUTPUT :
```

A lazy val is lazy until the first use

the RHS is delayed until the first use

then the variable is fixed to this value

This is useful for expensive resources

take a long time to calculate

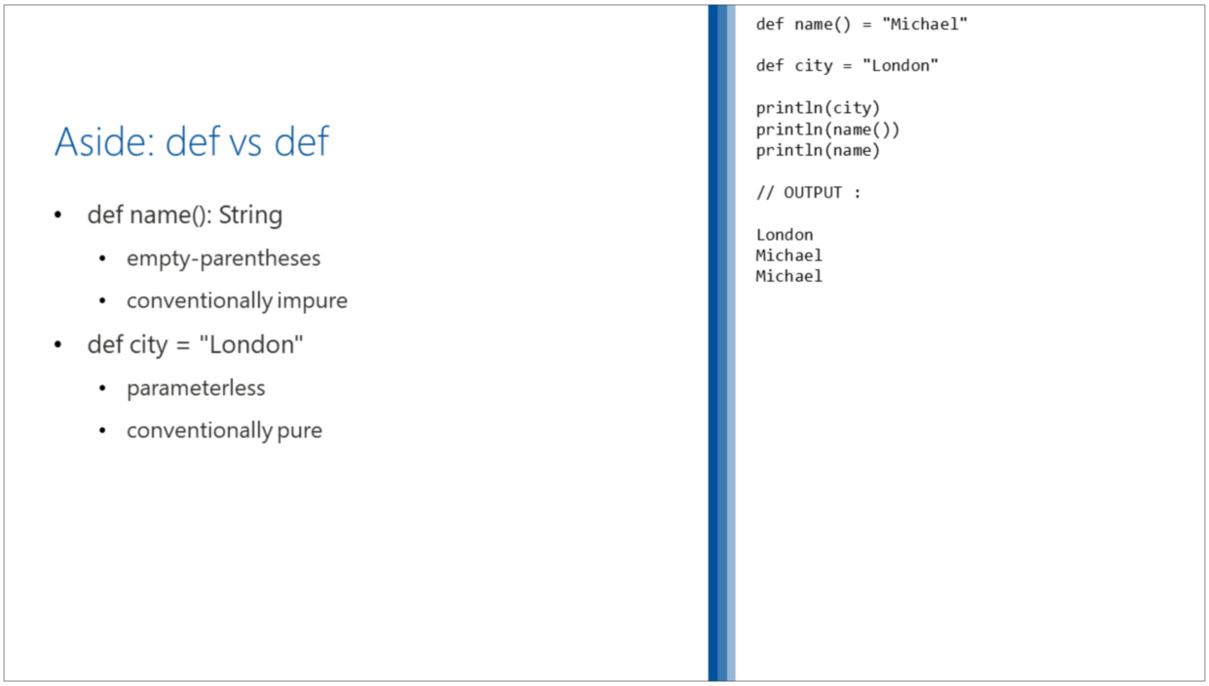
do not change after first calculated

possibly, may not be used in course of program

eg., a web page download

Lazy vals are also used for out-of-order definition

a lazy variable may refer to one defined later



Aside: def vs def

- def name(): String
  - empty-parentheses
  - conventionally impure
- def city = "London"
  - parameterless
  - conventionally pure

```
def name() = "Michael"

def city = "London"

println(city)
println(name())
println(name)

// OUTPUT :

London
Michael
Michael
```

def name(): String empty-paren method

def city = "London" is a parameterless method

convention is to use a parameterless method when pure

never define an impure (side-effectful) method as parameterless as it looks like field access

Exercise