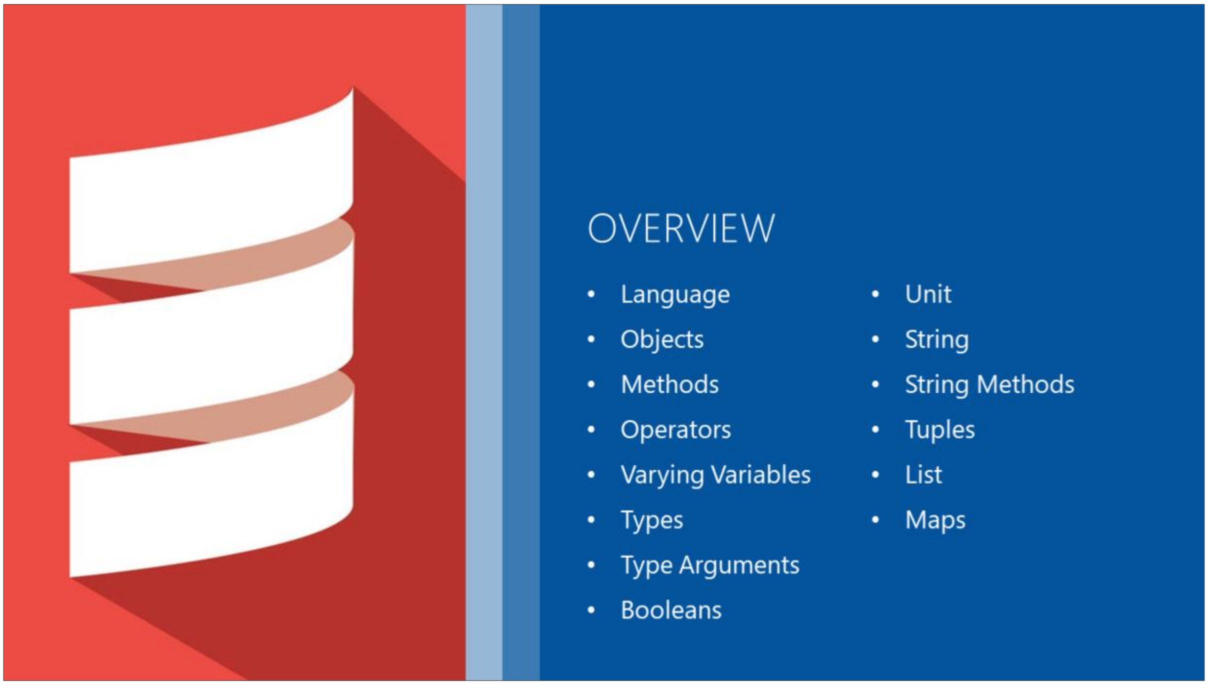
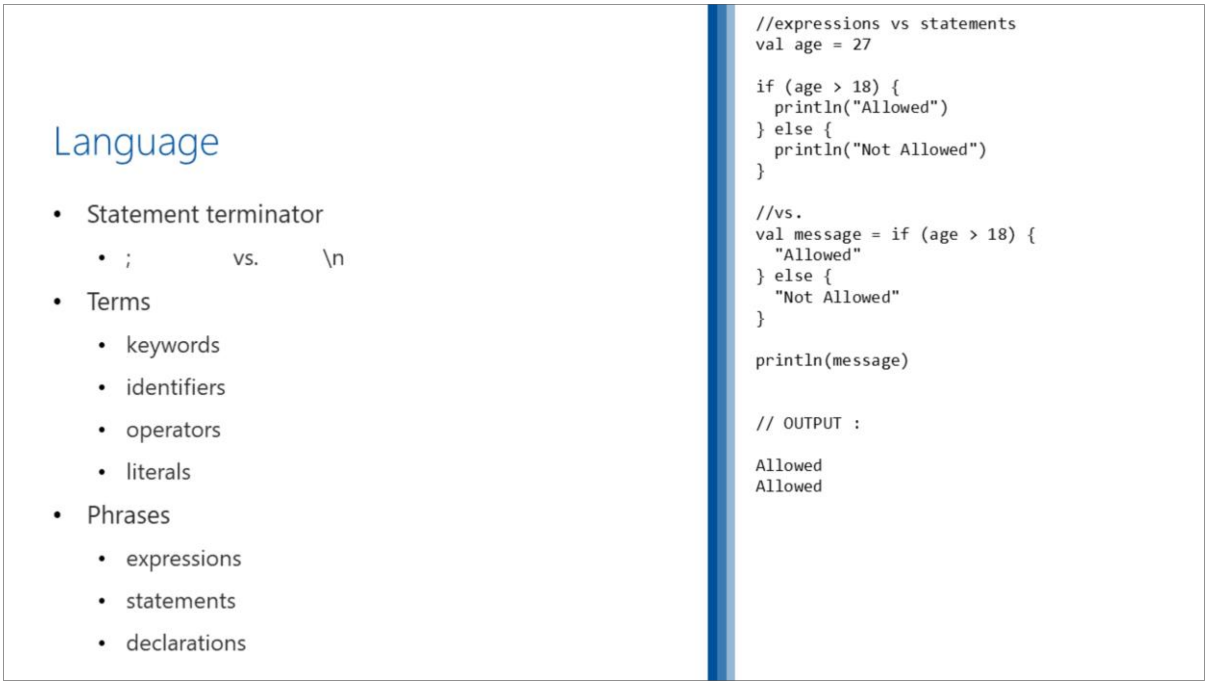
Fundamentals

SCALA PROGRAMMING
A DATA SCIENCE AND MACHINE LEARNING COURSE



OVERVIEW

- Language
- Objects
- Methods
- Operators
- Varying Variables
- Types
- Type Arguments
- Booleans
- Unit
- String
- String Methods
- Tuples
- List
- Maps



Language

- Statement terminator
  - ; vs. \n
- Terms
  - keywords
  - identifiers
  - operators
  - literals
- Phrases
  - expressions
  - statements
  - declarations

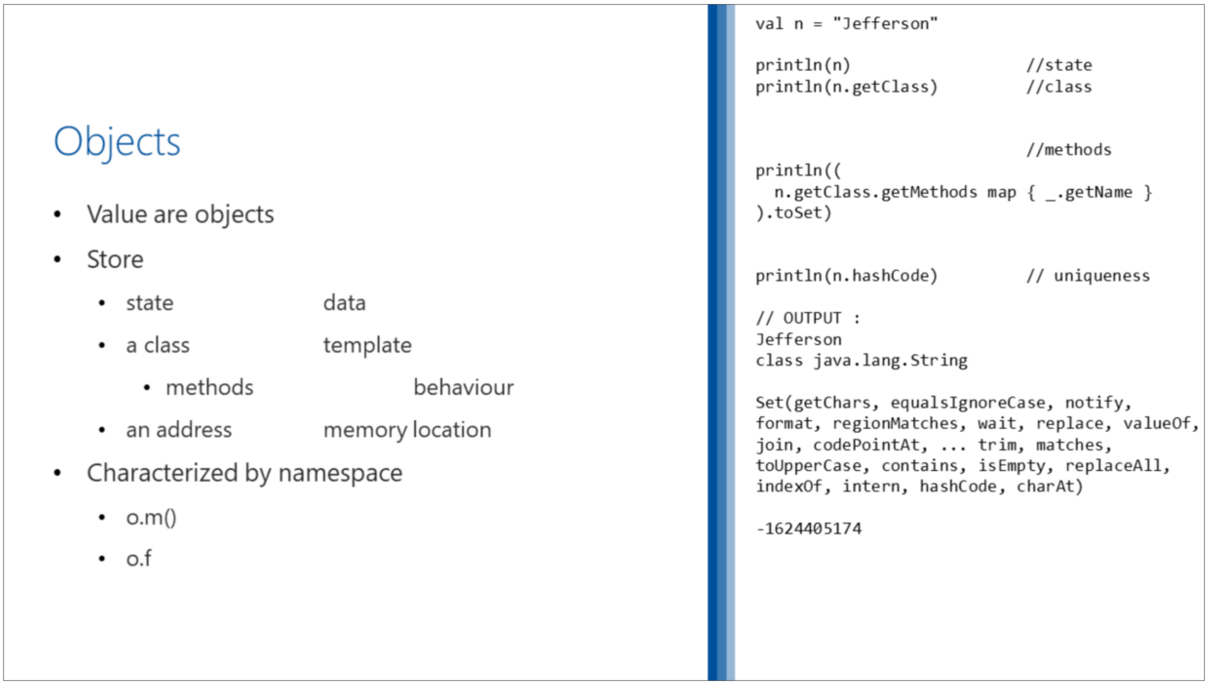
```
//expressions vs statements
val age = 27

if (age > 18) {
  println("Allowed")
} else {
  println("Not Allowed")
}

//vs.
val message = if (age > 18) {
  "Allowed"
} else {
  "Not Allowed"
}

println(message)


// OUTPUT :

Allowed
Allowed
```

Lines terminated with a new-line

or non-idiomatically, with a semi-colon

Scala syntax is mostly built from expressions

most phrases produce a value



Objects

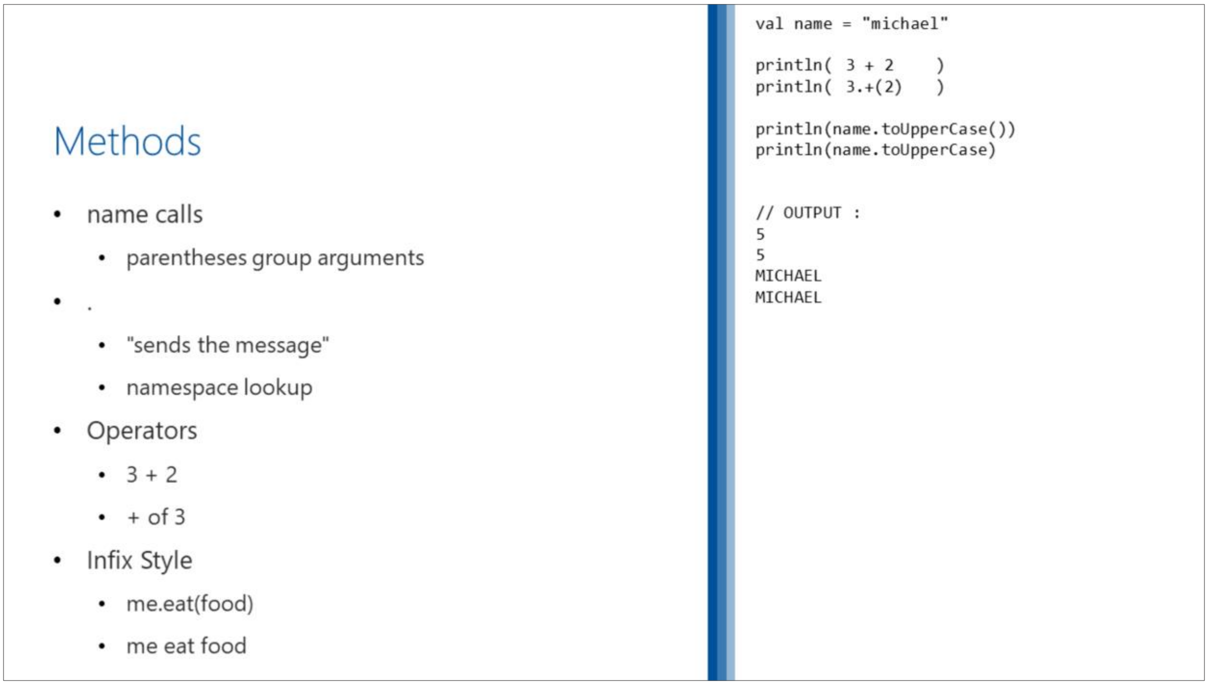
- Value are objects
- Store
  - state data
  - a class template
    - methods behaviour
  - an address memory location
- Characterized by namespace
  - o.m()
  - o.f

```
val n = "Jefferson"

println(n)                  //state
println(n.getClass)         //class


                            //methods
println((
  n.getClass.getMethods map { _.getName }
).toSet)


println(n.hashCode)         // uniqueness

// OUTPUT :
Jefferson
class java.lang.String

Set(getChars, equalsIgnoreCase, notify,
format, regionMatches, wait, replace, valueOf,
join, codePointAt, ... trim, matches,
toUpperCase, contains, isEmpty, replaceAll,
indexOf, intern, hashCode, charAt)

-1624405174
```

Every value is an object

An object is a data structure with

state	remembered data
a class	a remembered template
methods	behaviour
an address	a location in memory



Methods

- name calls
    - parentheses group arguments
- .
    - "sends the message"
    - namespace lookup
- Operators
    - 3 + 2
    - + of 3
- Infix Style
    - me.eat(food)
    - me eat food

```
val name = "michael"

println(  3 + 2     )
println(  3.+(2)    )

println(name.toUpperCase())
println(name.toUpperCase)


// OUTPUT :
5
5
MICHAEL
MICHAEL
```

Using a method name calls the method

parentheses group arguments

optional for zero or one arguments

. "sends the message"

i.e., calls the method

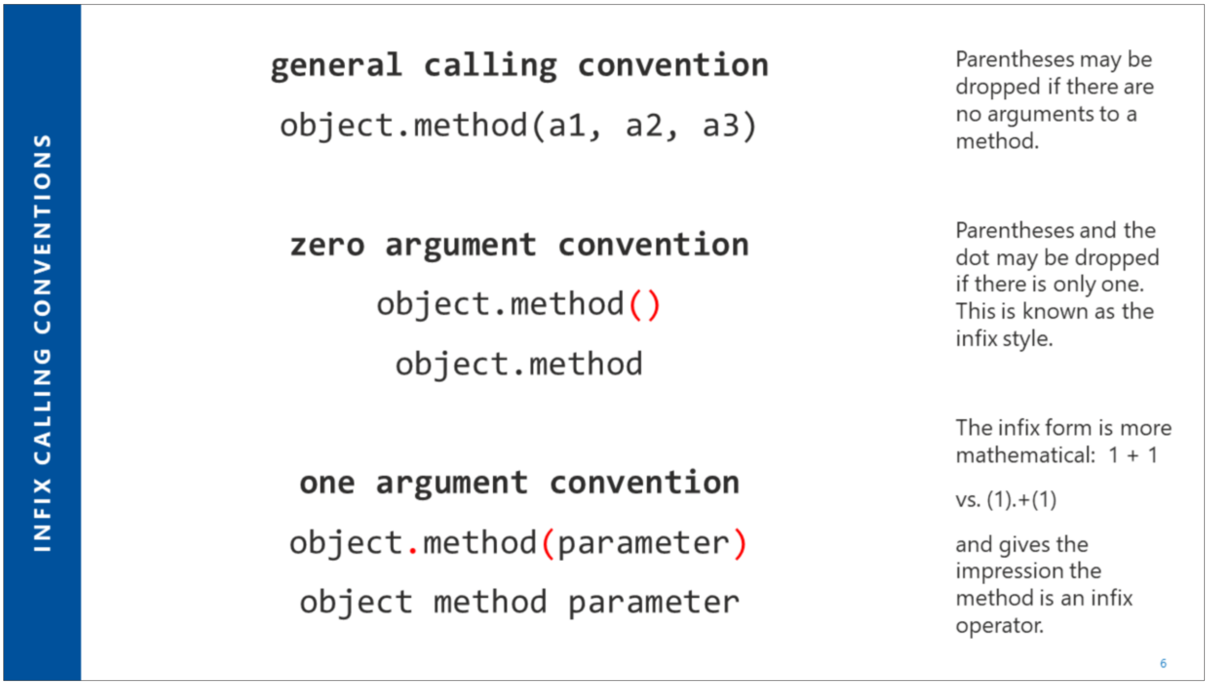
Operators are methods

3 + 2 is a method call on the object 3

the method is named +

Methods of one argument may be called without .

me.eat(food) is the same as me eat food



INFIX CALLING CONVENTIONS

general calling convention

```
object.method(a1, a2, a3)
```

Parentheses may be dropped if there are no arguments to a method.

zero argument convention

```
object.method()
object.method
```

Parentheses and the dot may be dropped if there is only one. This is known as the infix style.

one argument convention

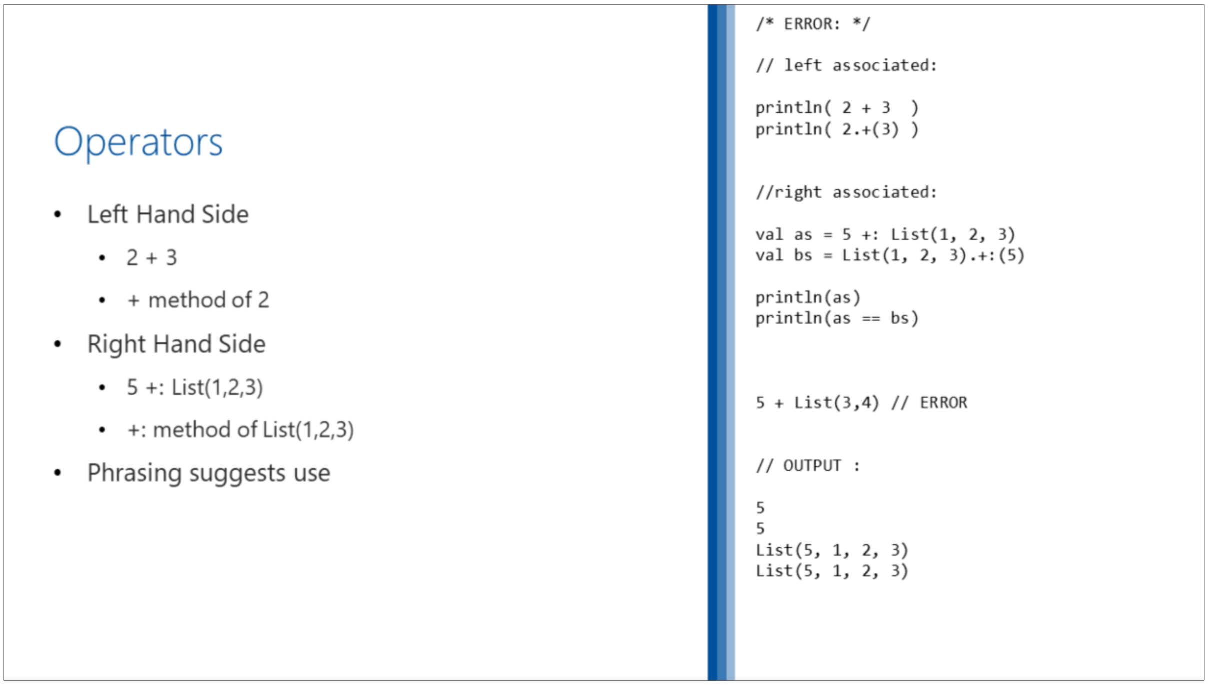
```
object.method(parameter)
object method parameter
```

The infix form is more mathematical: 1 + 1

vs. (1).+(1)

and gives the impression the method is an infix operator.

6



Operators

- Left Hand Side
  - 2 + 3
  - + method of 2
- Right Hand Side
  - 5 +: List(1,2,3)
  - +: method of List(1,2,3)
- Phrasing suggests use

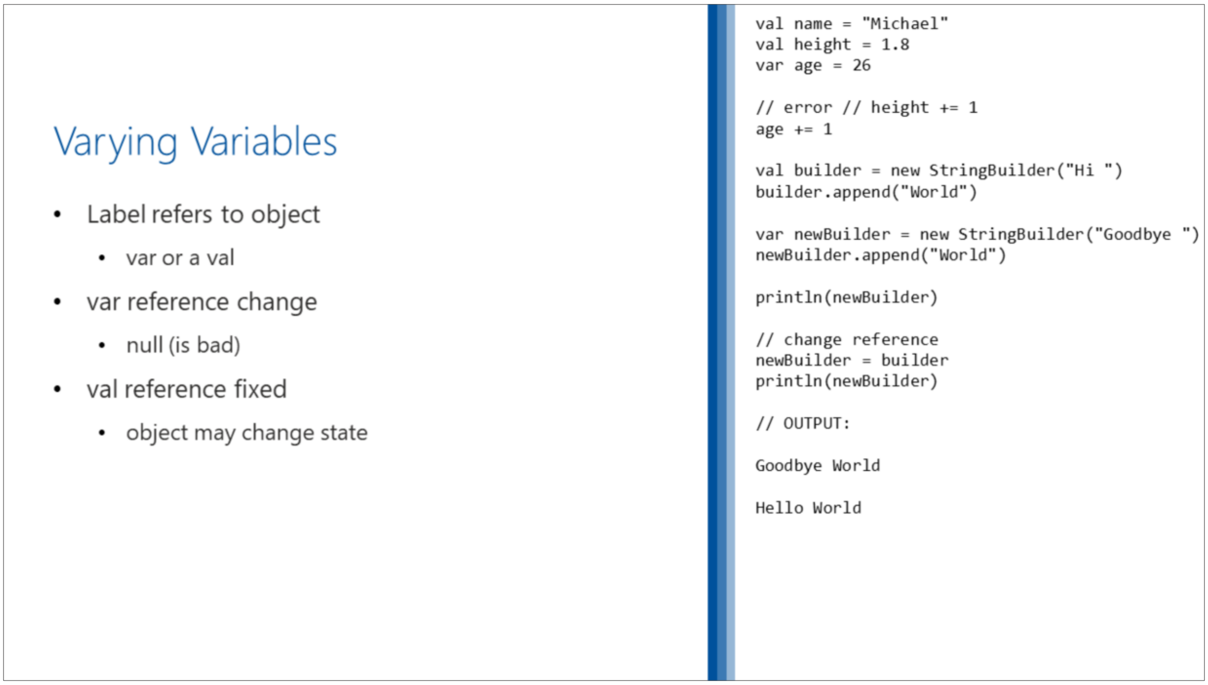
```
/* ERROR: */

// left associated:

println( 2 + 3  )
println( 2.+(3) )


//right associated:

val as = 5 +: List(1, 2, 3)
val bs = List(1, 2, 3).+:(5)

println(as)
println(as == bs)



5 + List(3,4) // ERROR


// OUTPUT :

5
5
List(5, 1, 2, 3)
List(5, 1, 2, 3)
```

Operators are called on the left-hand-side object

2 + 3 is the + method of 2

However operators with a : in their name are called on the right-hand side object

5 +: List(1,2,3) is the +: method of List(1,2,3)



Varying Variables

- Label refers to object
  - var or a val
- var reference change
  - null (is bad)
- val reference fixed
  - object may change state

```
val name = "Michael"
val height = 1.8
var age = 26

// error // height += 1
age += 1

val builder = new StringBuilder("Hi ")
builder.append("World")

var newBuilder = new StringBuilder("Goodbye ")
newBuilder.append("World")

println(newBuilder)

// change reference
newBuilder = builder
println(newBuilder)

// OUTPUT:

Goodbye World

Hello World
```

An identifier which labels an object is a reference

An identifier may be a var or a val

A var identifier can change which object it refers to

may refer to null

in scala, null references are bad practice

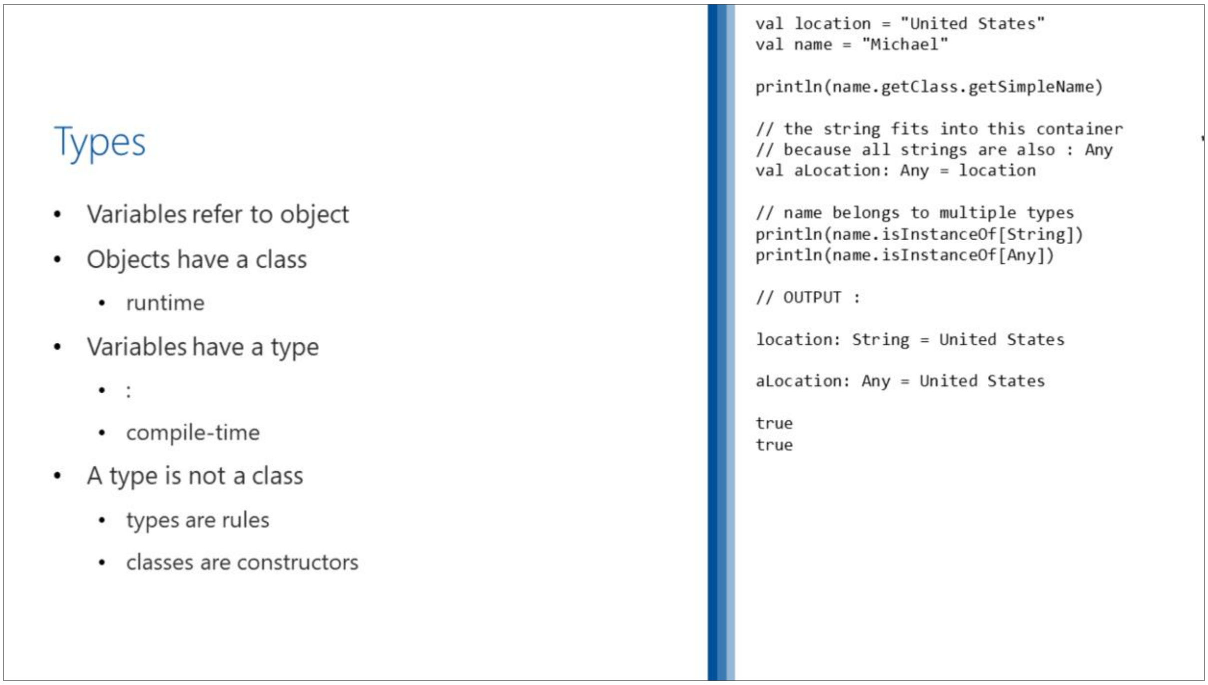
A val identifier cannot change which object it refers to

however the object itself may change its state

The state of an object is unaffected by var/val

these apply to identifiers not to memory

Both var/val are known as variables



Types

- Variables refer to object
- Objects have a class
  - runtime
- Variables have a type
  - :
  - compile-time
- A type is not a class
  - types are rules
  - classes are constructors

```
val location = "United States"
val name = "Michael"

println(name.getClass.getSimpleName)

// the string fits into this container
// because all strings are also : Any
val aLocation: Any = location

// name belongs to multiple types
println(name.isInstanceOf[String])
println(name.isInstanceOf[Any])

// OUTPUT :

location: String = United States

aLocation: Any = United States

true
true
```

Every variable refers to an object

Every object has a class

A class is an in-memory runtime association

The in-memory object knows which class it belongs to

Every variable has a type

given after a :

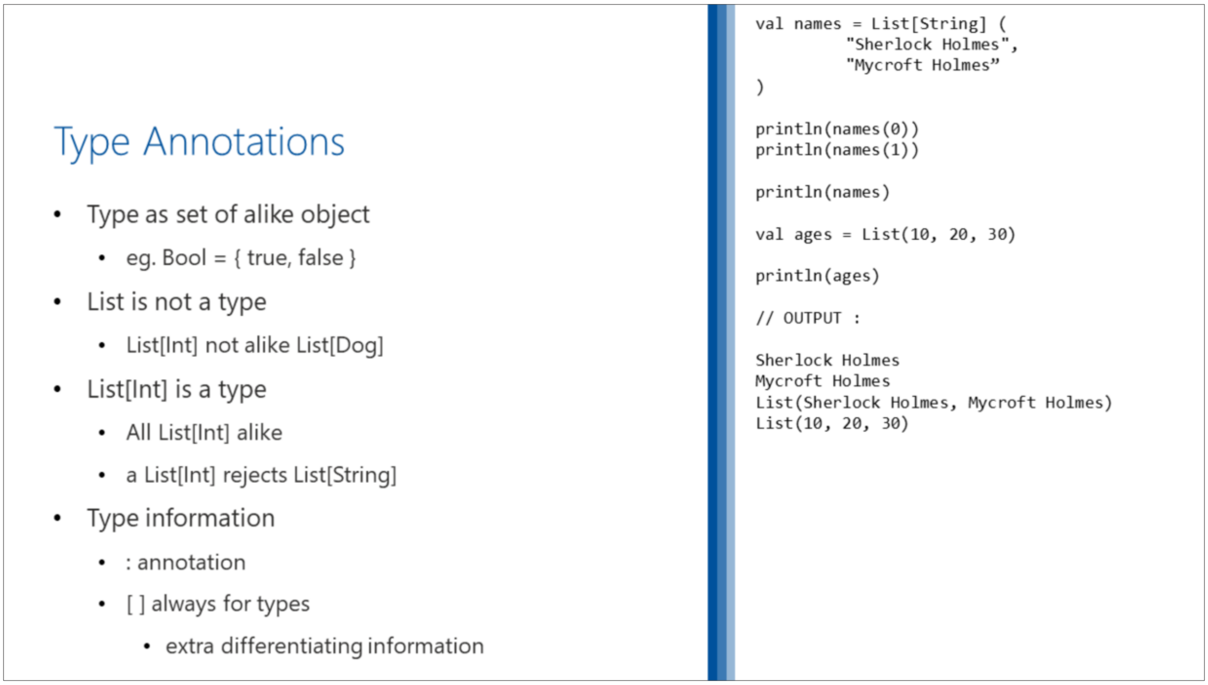
describes at compile-time which objects it may refers to

A type is not a class

The type system is a compile time rule set

Applies to terms in the program source

Memory and therefore objects have no type



Type Annotations

- Type as set of alike object
  - eg. Bool = { true, false }
- List is not a type
  - List[Int] not alike List[Dog]
- List[Int] is a type
  - All List[Int] alike
  - a List[Int] rejects List[String]
- Type information
  - : annotation
  - [ ] always for types
    - extra differentiating information

```
val names = List[String] (
         "Sherlock Holmes",
         "Mycroft Holmes”
)

println(names(0))
println(names(1))

println(names)

val ages = List(10, 20, 30)

println(ages)

// OUTPUT :

Sherlock Holmes
Mycroft Holmes
List(Sherlock Holmes, Mycroft Holmes)
List(10, 20, 30)
```

To simplify, a type can be considered a set

a group of allowable objects

The type Bool is therefore a set of two values

true, false

List is not a type

a list of dogs is not the same as a list of ingredients

more information is needed to know what is allowable

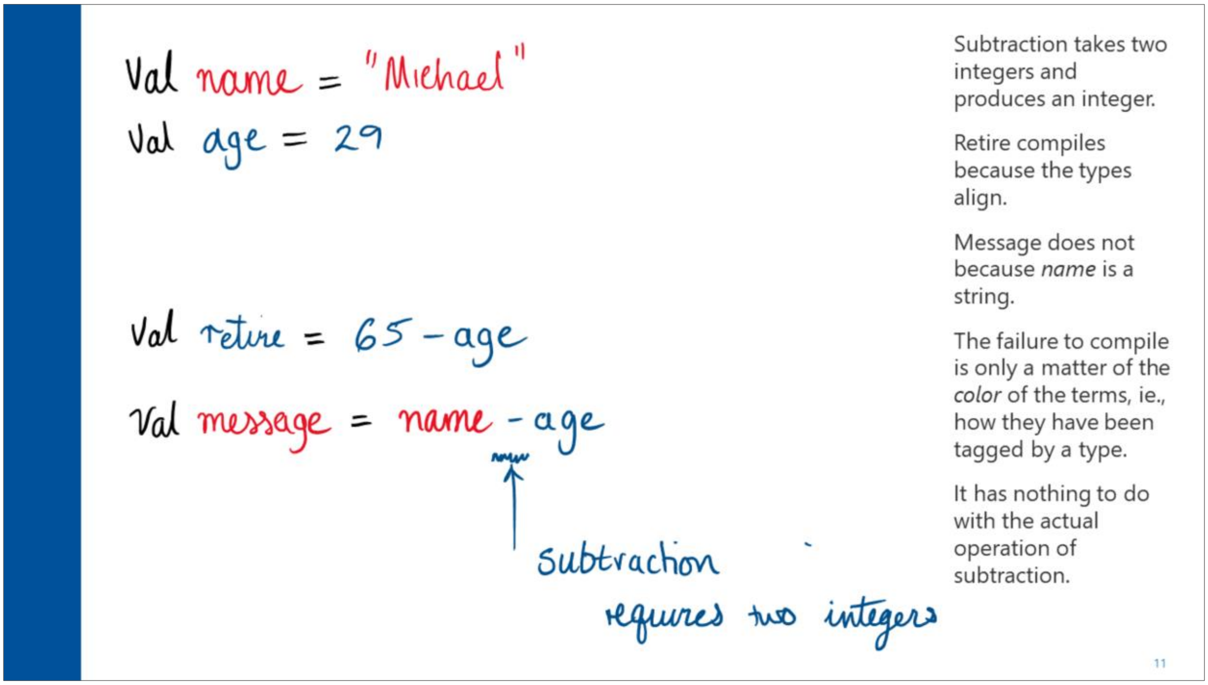
List[Int] is a type

it is all the information needed to know what is allowed

a List[Int] is rejected when a List[String] is required

In scala, square brackets are always type information

In F[A] , A can be considered a argument to F



```
Val name = "Michael"
Val age = 29

Val retire = 65 - age

Val message = name - age
```

subtraction

requires two integers

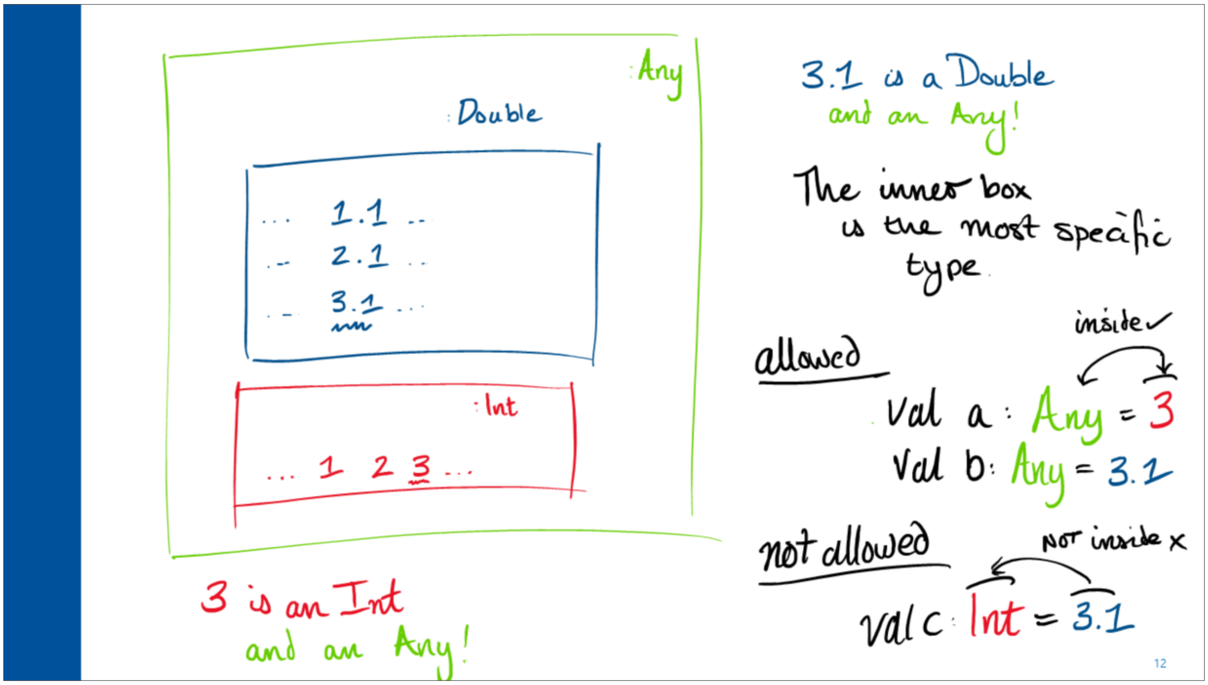
Subtraction takes two integers and produces an integer.

Retire compiles because the types align.

Message does not because *name* is a string.

The failure to compile is only a matter of the *color* of the terms, ie., how they have been tagged by a type.

It has nothing to do with the actual operation of subtraction.



3 is an Int
and an Any!

3.1 is a Double
and an Any!

The inner box
is the most specific
type.

allowed

inside ✓

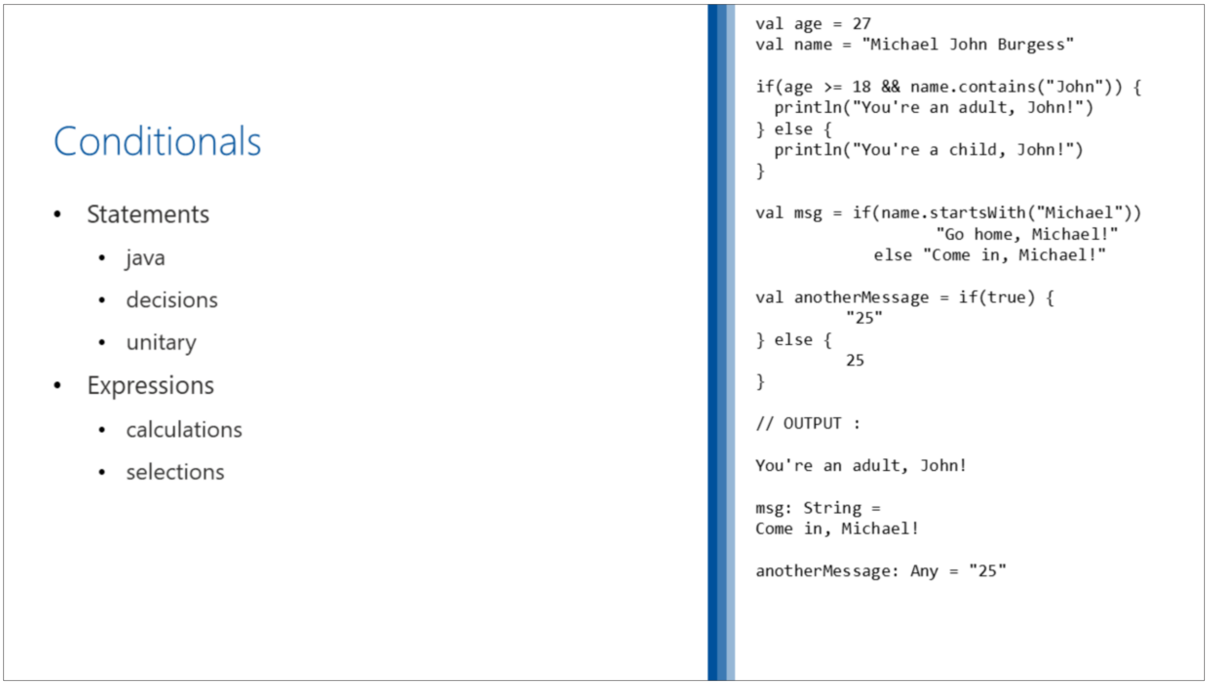
val a : Any = 3

val b: Any = 3.1

not allowed

NOT inside ×

val c: Int = 3.1



Conditionals

- Statements
  - java
  - decisions
  - unitary
- Expressions
  - calculations
  - selections

```
val age = 27
val name = "Michael John Burgess"

if(age >= 18 && name.contains("John")) {
  println("You're an adult, John!")
} else {
  println("You're a child, John!")
}

val msg = if(name.startsWith("Michael"))
                   "Go home, Michael!"
             else "Come in, Michael!"

val anotherMessage = if(true) {
         "25"
} else {
         25
}

// OUTPUT :

You're an adult, John!

msg: String =
Come in, Michael!

anotherMessage: Any = "25"
```

In imperative language conditionals are used to make decisions

If some conditions is met, perform some action

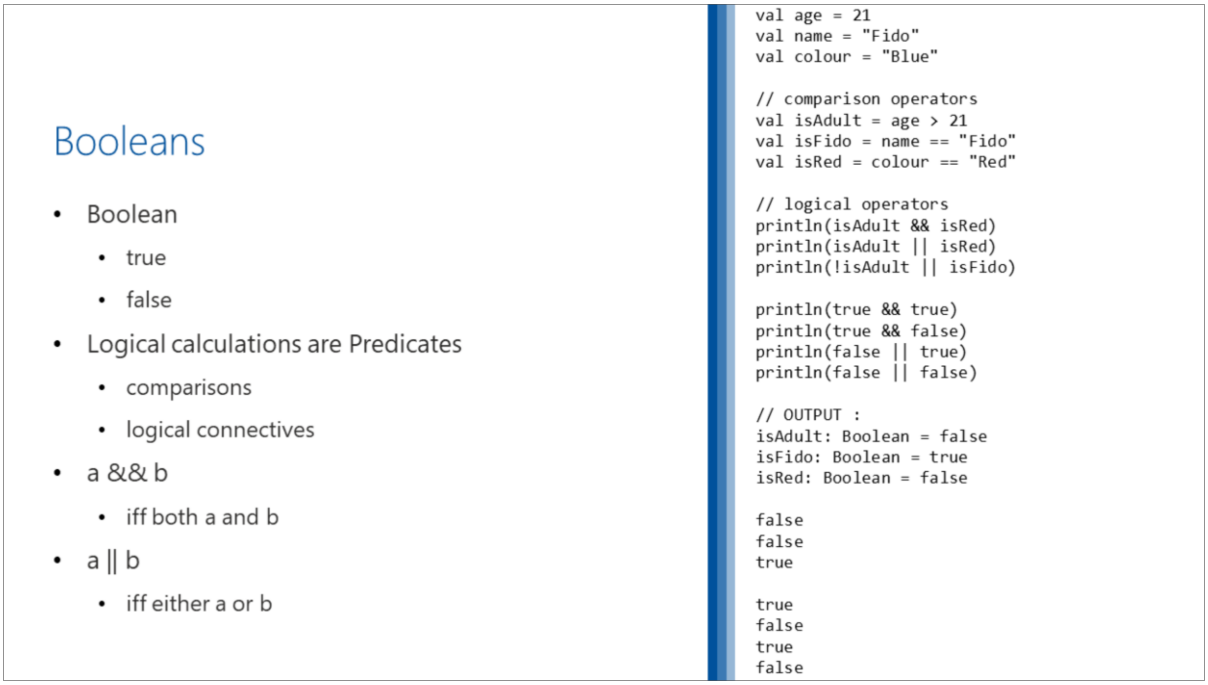
In scala, actions are to be resisted

Conditional constructs are used for selecting values

Each "branch" of the if/else if/else must contain a value

which may be Unit

the whole expression is evaluated to the value of the true-branch



Booleans

- Boolean
  - true
  - false
- Logical calculations are Predicates
  - comparisons
  - logical connectives
- a && b
  - iff both a and b
- a || b
  - iff either a or b

```
val age = 21
val name = "Fido"
val colour = "Blue"

// comparison operators
val isAdult = age > 21
val isFido = name == "Fido"
val isRed = colour == "Red"

// logical operators
println(isAdult && isRed)
println(isAdult || isRed)
println(!isAdult || isFido)

println(true && true)
println(true && false)
println(false || true)
println(false || false)

// OUTPUT :
isAdult: Boolean = false
isFido: Boolean = true
isRed: Boolean = false

false
false
true

true
false
true
false
```

There are two values which belong to the type boolean

true

false

True is the value of logical calculations, or predicates, which describe facts

predicates are composed of comparisons and tests (eg. name == "Fido")

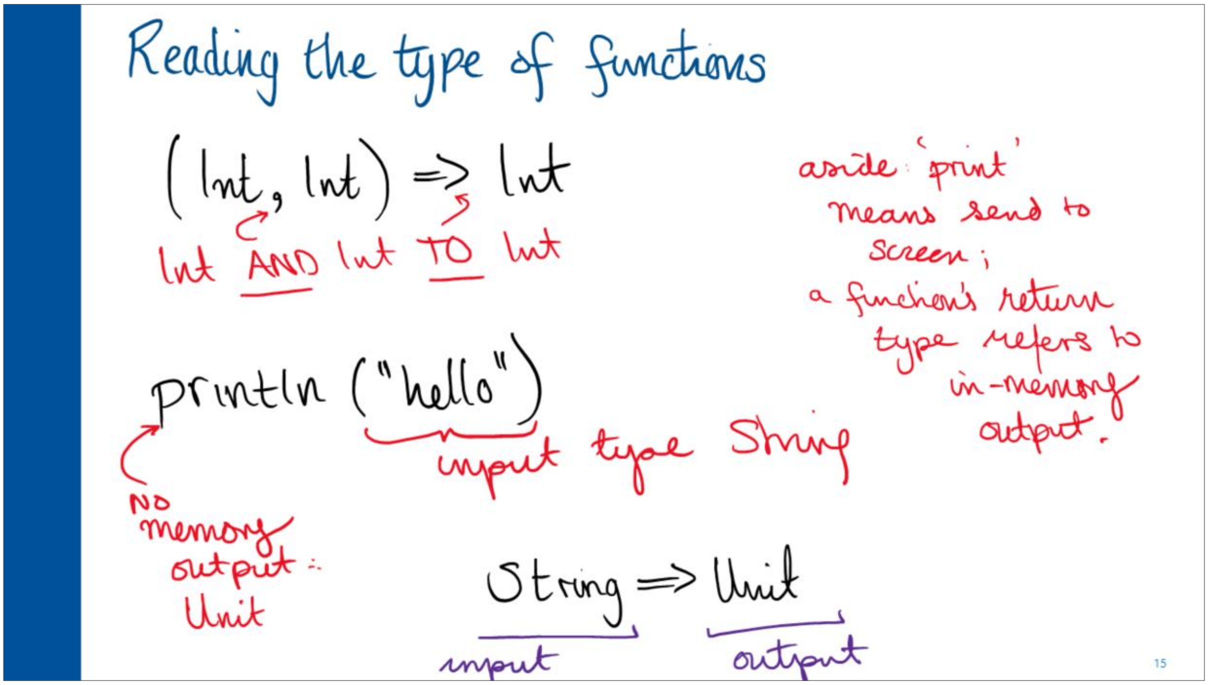
and logical connectives which determine how these tests will be connected

a && b is true iff both a is true and b are true, otherwise false

a || b is true iff either a or b is true, otherwise false

Expressions such as (a == b) && (c != d) are calculations

they have a boolean value, being either true or false



Reading the type of functions

(Int, Int) => Int

Int AND Int TO Int

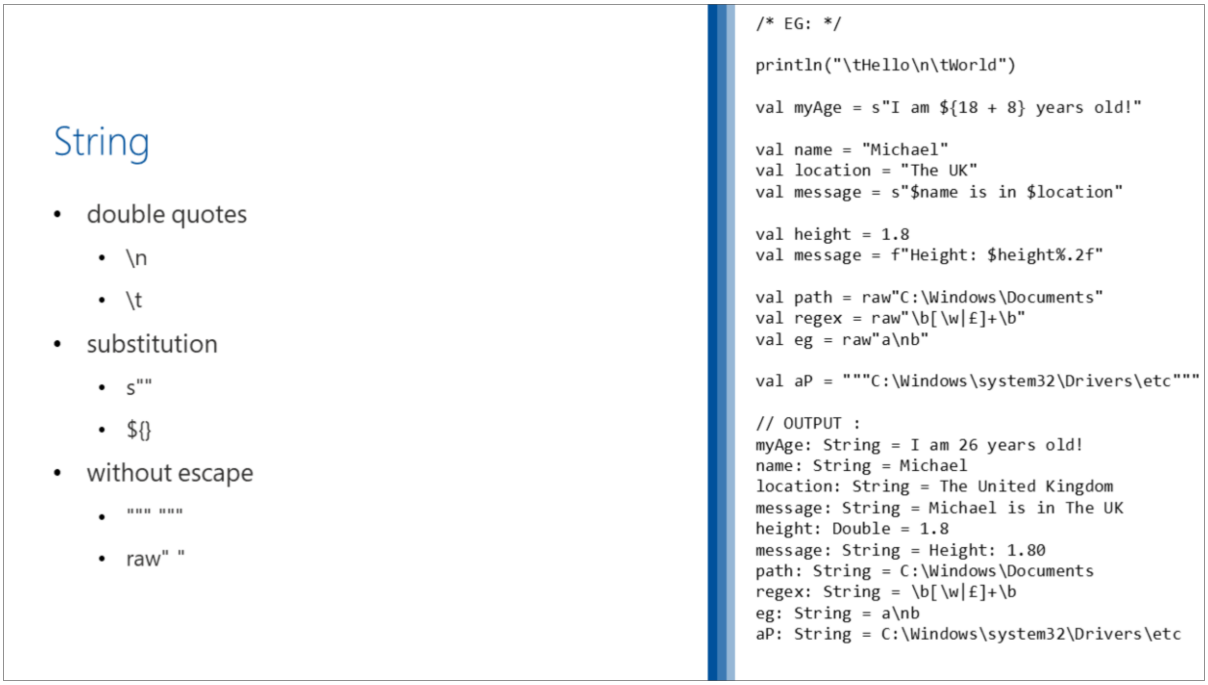
aside: 'print' means send to screen; a function's return type refers to in-memory output.

$\text{println}\ \underbrace{(\text{"hello"})}_{\text{input type String}}$

NO memory output ∴ Unit

$$\underbrace{\text{String}}_{\text{input}} \Rightarrow \underbrace{\text{Unit}}_{\text{output}}$$

15



String

- double quotes
  - \n
  - \t
- substitution
  - s""
  - ${}
- without escape
  - """" """"
  - raw" "

```
/* EG: */

println("\tHello\n\tWorld")

val myAge = s"I am ${18 + 8} years old!"

val name = "Michael"
val location = "The UK"
val message = s"$name is in $location"

val height = 1.8
val message = f"Height: $height%.2f"

val path = raw"C:\Windows\Documents"
val regex = raw"\b[\w|£]+\b"
val eg = raw"a\nb"

val aP = """C:\Windows\system32\Drivers\etc"""

// OUTPUT :
myAge: String = I am 26 years old!
name: String = Michael
location: String = The United Kingdom
message: String = Michael is in The UK
height: Double = 1.8
message: String = Height: 1.80
path: String = C:\Windows\Documents
regex: String = \b[\w|£]+\b
eg: String = a\nb
aP: String = C:\Windows\system32\Drivers\etc
```

Strings are defined with double quotes

Substitute escape characters by default

Eg., \n becomes a newline, \t a tab

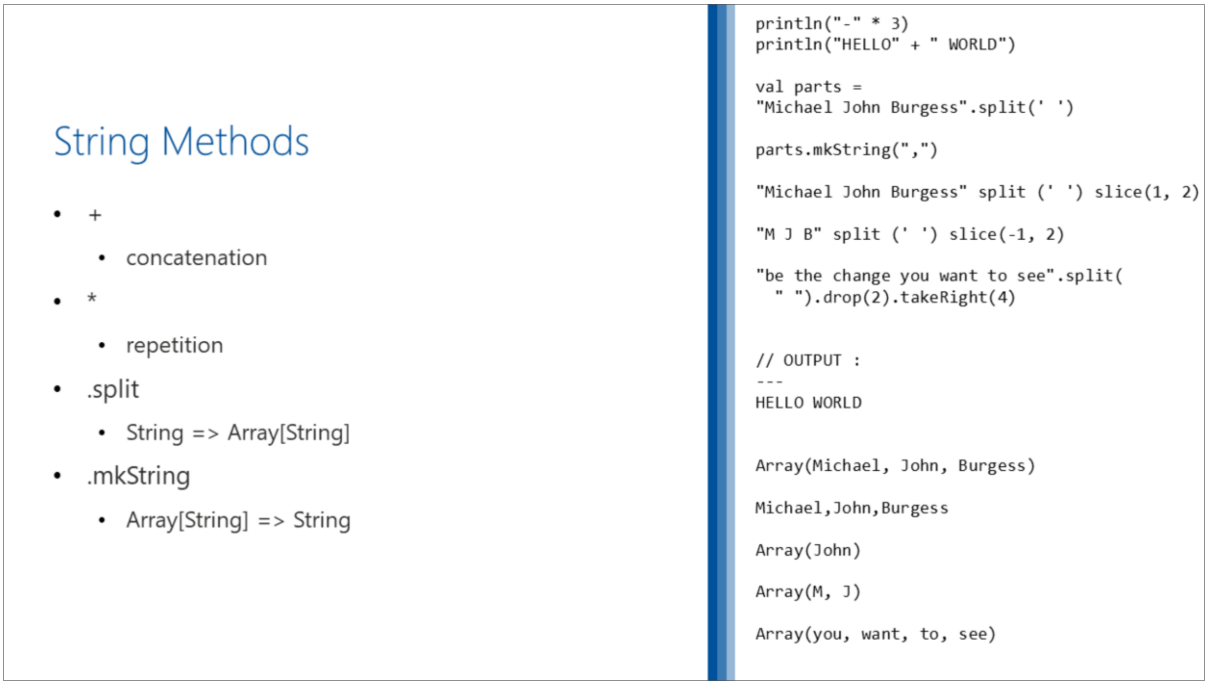
With an s prefix substitute expression formatted with ${}

With a raw prefix ignores escapes

\n remains \n

With triple double quotes behave as raw strings

Raw strings are especially useful for windows file paths and regular expressions where \ has a specific meaning



String Methods

- +
  - concatenation
- *
  - repetition
- .split
  - String => Array[String]
- .mkString
  - Array[String] => String

```
println("-" * 3)
println("HELLO" + " WORLD")

val parts =
"Michael John Burgess".split(' ')

parts.mkString(",")

"Michael John Burgess" split (' ') slice(1, 2)

"M J B" split (' ') slice(-1, 2)

"be the change you want to see".split(
  " ").drop(2).takeRight(4)


// OUTPUT :
---
HELLO WORLD


Array(Michael, John, Burgess)

Michael,John,Burgess

Array(John)

Array(M, J)

Array(you, want, to, see)
```

Operators:

+ for concatenation

* for repetition

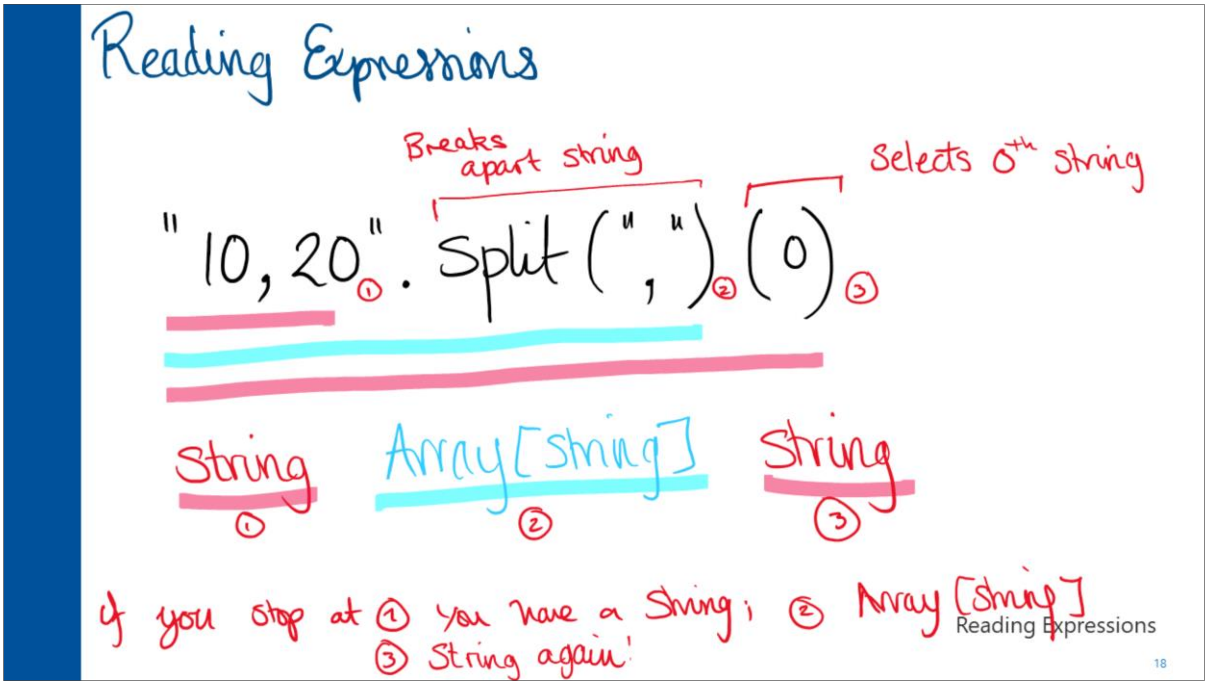
Named methods:

.split divides a string into an Array[String]

.mkString on an Array[String] glues an array back to a String

Note that since .split takes one argument it can be called in infix style:

("UK, London" split ", ") == Array("UK", "London")



Reading Expressions

Breaks apart string

Selects 0th string

"10, 20"① . Split(",")② (0)③

String ①

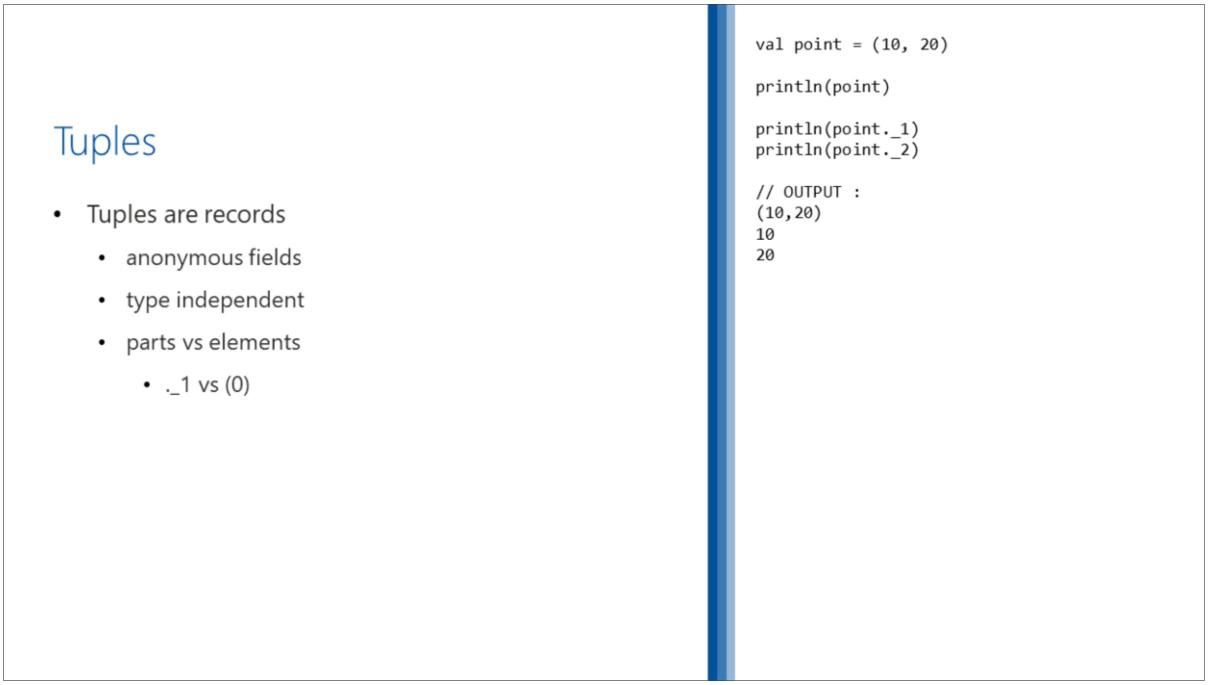
Array[String] ②

String ③

If you stop at ① you have a String; ② Array[String] ③ String again!

Reading Expressions

18



Tuples

- Tuples are records
  - anonymous fields
  - type independent
  - parts vs elements
    - ._1 vs (0)

```
val point = (10, 20)

println(point)

println(point._1)
println(point._2)

// OUTPUT :
(10,20)
10
20
```

Tuples are analogues to structures, class-type objects or "records"

Each field of a tuple is named anonymously

And has a type independent of the other fields

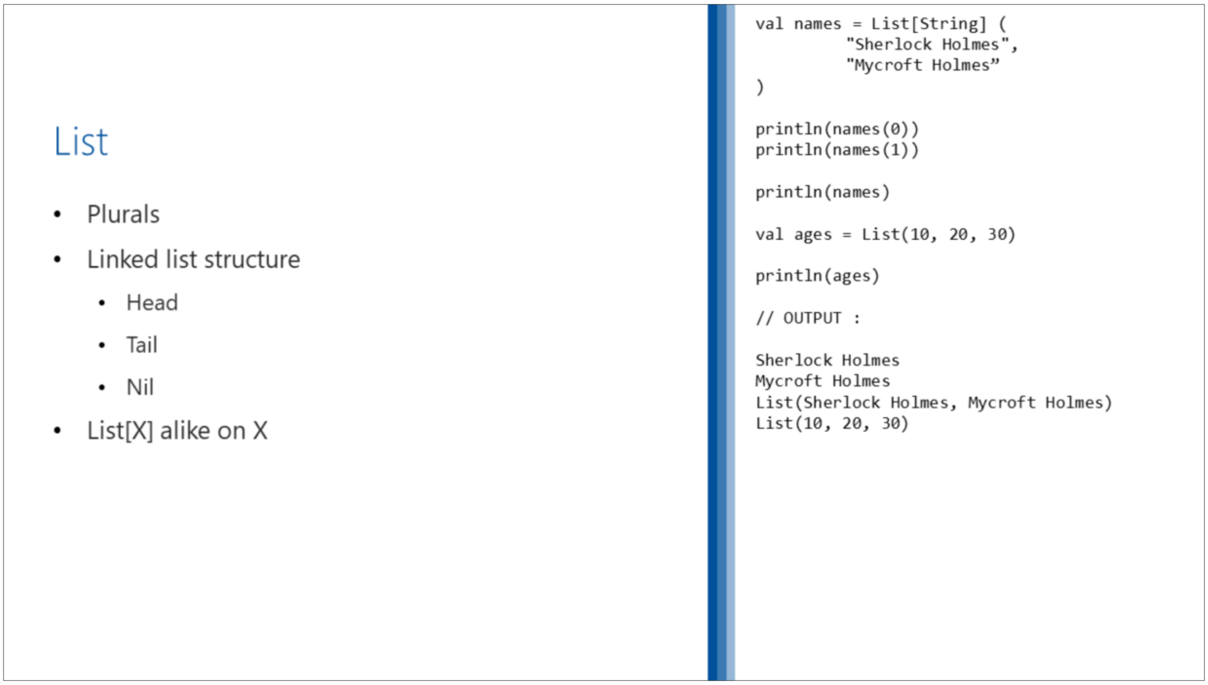
A tuple is therefore heterogenous

It has parts of a different type

Unlike collections

Tuples cannot be indexed as the indexer would have to return Any to be compatible with all field types

aside: the .productIterator method provides an alternative but almost always a collections is better



List

- Plurals
- Linked list structure
  - Head
  - Tail
  - Nil
- List[X] alike on X

```
val names = List[String] (
         "Sherlock Holmes",
         "Mycroft Holmes"
)

println(names(0))
println(names(1))

println(names)

val ages = List(10, 20, 30)

println(ages)

// OUTPUT :

Sherlock Holmes
Mycroft Holmes
List(Sherlock Holmes, Mycroft Holmes)
List(10, 20, 30)
```

Lists conveniently represent plural or grouped data

Lists are linked list data structures:

A head element connected to a List

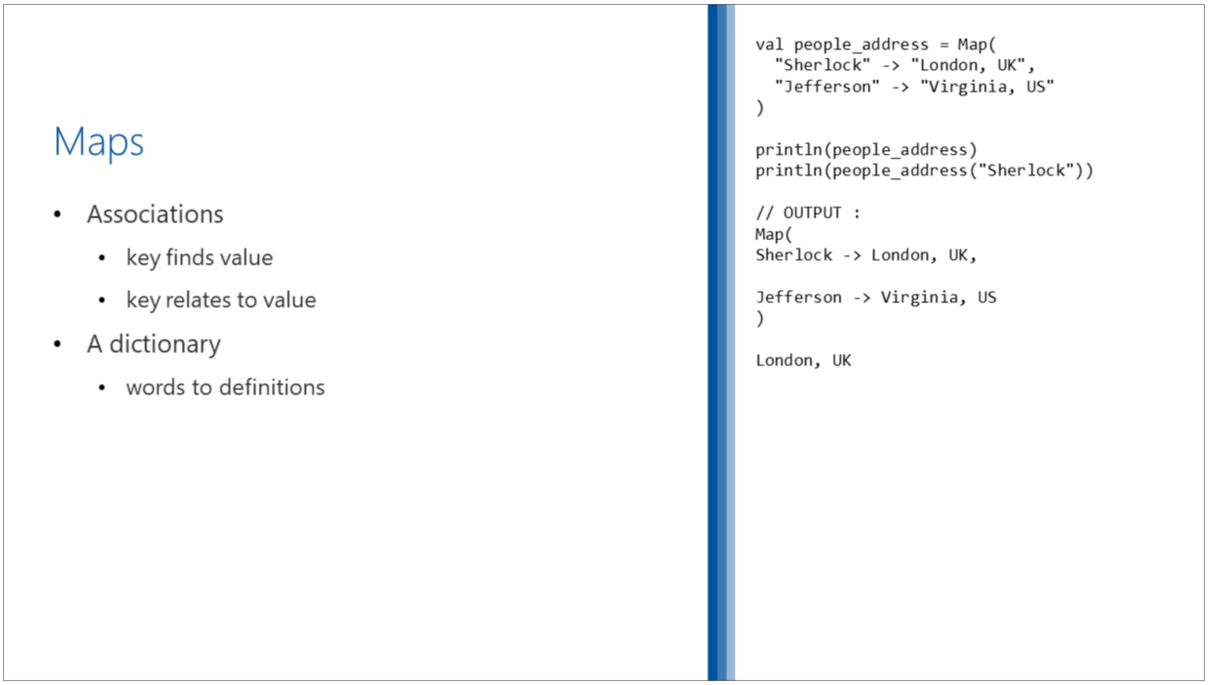
This list has a element and is connected to list

All the way to an empty List

In general F[A] just means an F[A] is an F[B] iff A is B, ie., that A discriminates between varieties of F.

Lists, and most data structures, are disciminated by element type.

So List[Int] means roughly, “A list with integer elements”



Maps

- Associations
  - key finds value
  - key relates to value
- A dictionary
  - words to definitions

```
val people_address = Map(
  "Sherlock" -> "London, UK",
  "Jefferson" -> "Virginia, US"
)

println(people_address)
println(people_address("Sherlock"))

// OUTPUT :
Map(
Sherlock -> London, UK,

Jefferson -> Virginia, US
)

London, UK
```

Maps are associations

collections of key-value pairs

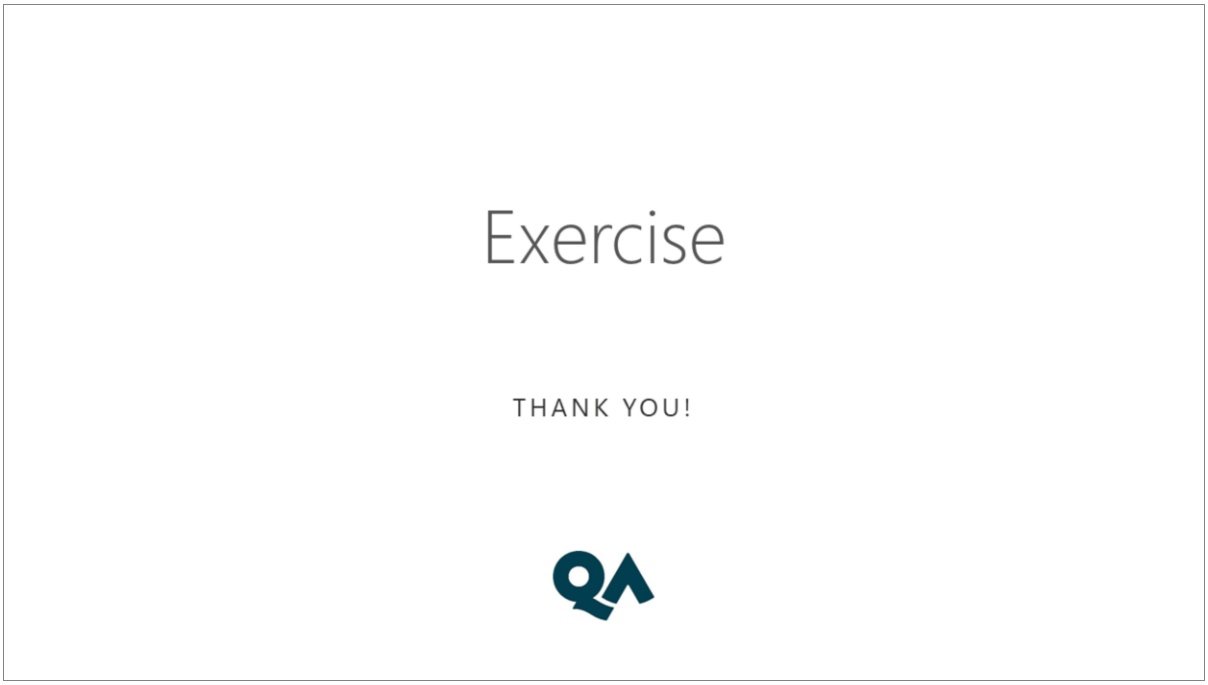
the key may be used to lookup the value

A simple example of a Map is a dictionary

associates words to their definitions

the key is the word

the definition is the value



Exercise

THANK YOU!