# PROGRAMMING IN SCALA

Scala Basics

# Define some variables

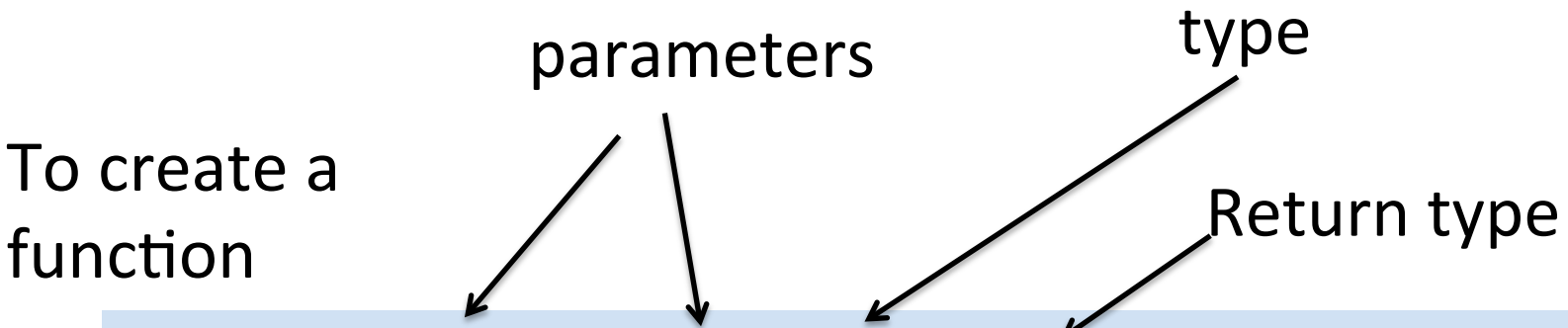
two kinds of variables `vals` and `vars`
`val` is like a final variable in Java
once initialized, cannot be re-assigned

```
val myList = List(1,2,3,4,5)
val empty: List[Nothing] = List()

val msg = “Hello World”
val msg2: String = “Hello again”

var greeting = “Hello World”
greeting = “leave me alone”
```

# Define some functions

- function parameters must have type annotations
- Scala compiler does not infer function parameter types
- specifying the result type is sometimes optional
- if the function consists of just one statement, braces can be left out

# Define some functions



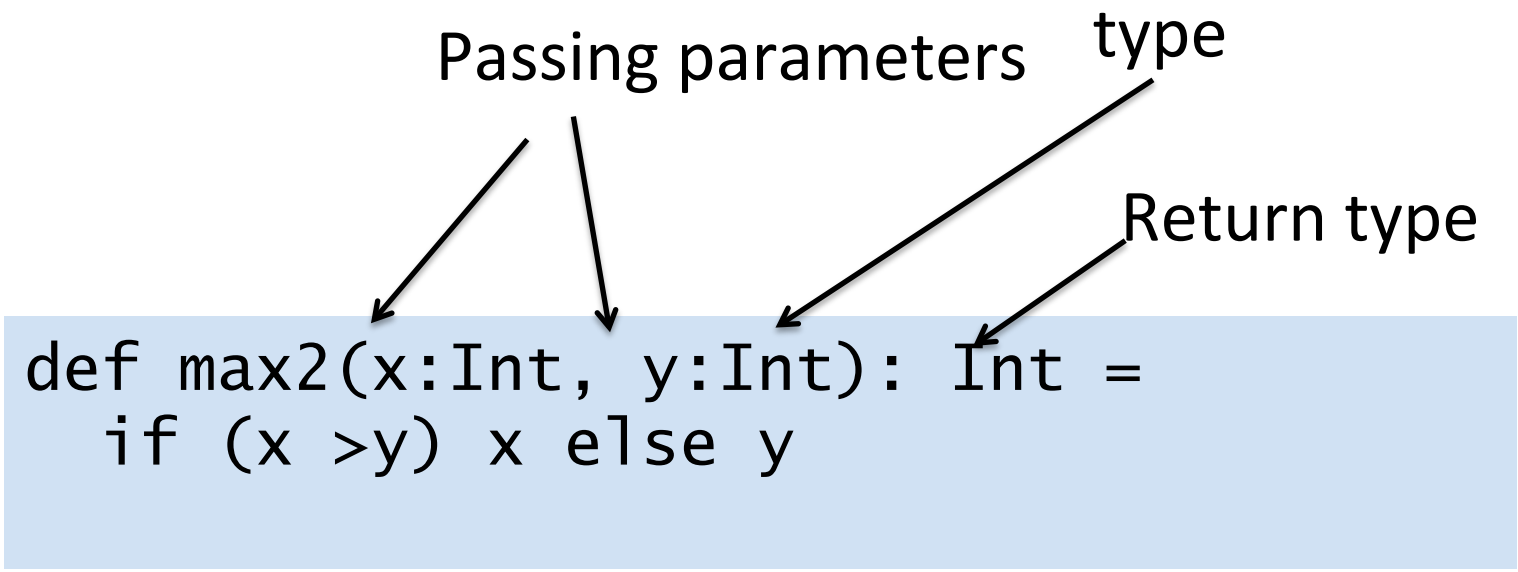

```
def max(x:Int, y:Int): Int = {
  if (x >y) x
  else y
}
```

To call a function ➔ name and value

```
max(20,40)
```

# Define some functions

## another way



# Define some functions

- a function that takes no parameters and returns no interesting result
- result type of `greet` is **`Unit`**
- **`Unit`** is similar (but not exactly the same as!) to Java's `void` type
- a result type of **`Unit`** is an indicator that a function has side effects, in fact, only executed for its side effects

```
def greet() = println("Hello World")

def greet(): Unit = println("Hello World")
```

Output

```
<console>:11: warning: enclosing method greet has result
type Unit: return value discarded
       def greet(x:Int, y:Int): Unit ={ println("Hello
World"); return x}
```

# Comments

Commenting is just like Java

- compiler ignores characters between // and end of line
- ignores characters between /* and */

```
// this is comment line
def greet() = println("Hello World")

/* block comment
def greet(): Unit = println("Hello World")
*/
```

# while, if

**while** loops and **if** statement are just similar to what you would see in a Java program.

```
val myStrings = Array("zero", "one", "two")
var i = 0
   while (i < myStrings.length) {
     println(myStrings(i))
         i += 1
   }
```

# If statements

```
val myStrings = Array("zero", "one", "two")
var i = 0
    while (i < myStrings.length) {
            if (i != 0){
               print(" ")
            }
           print(myStrings(i))
               i += 1
     }
```

# Iterate with **for**

Here, **aString** is the name of a val, not a var
**aString** gets a new value on each iteration, but really is a val, can't be reassigned inside the body of the for expression

```
val myStrings = Array("zero", "one", "two")

for (aString <- myStrings)
  println(aString)
```

# Iterate with **foreach**, **for**

**foreach** method takes a function and applies it to each item in a sequence

```
val myStrings = Array("zero", "one", "two")

// functions are firstclass objects
myStrings.foreach(aString => println(aString))

// with type annotation
myStrings.foreach((aString: String) => println(aString))

// single argument function, leave out the argument
myStrings.foreach(println)
```

# Iterate with **foreach**, **for**

**foreach** method takes a function and applies it to each item in a sequence

```
val myStrings = Array("1", "2", "3")

// with type annotation (casting type)
myStrings.foreach((aString: Int) => println(aString*aString))
```

# Arrays

Just like Java:

- Arrays are **homogenous** (all elements in an array have the same type.)

```
val myGreetings = new Array[String](3)

myGreetings(0) = "Hello"
myGreetings(1) = ", "
myGreetings(2) = "world!\n"

for (i <- 0 to 2)
  print(myGreetings(i))
```

Array of a size 3

Assigning values to the array

# Arrays

Just like Java:

- Arrays are **mutable**, individual elements can be updated.

```
val myGreetings = new Array[String](3)

myGreetings(0) = "Hello"
myGreetings(1) = ", "
myGreetings(2) = "world!\n"
myGreetings(2) = ”Mr!\n"

for (i <- 0 to 2)
  print(myGreetings(i))
```

Array of a size 3

Values of elements can be changed

# Arrays are mutable

Arrays are homogenous (all elements in an array have the same type.)
Arrays are mutable objects, individual elements can be updated.

```
val myGreetings = new Array[String](3)

myGreeetings(0) = “Hello”
myGreetings(1) = “, ”
myGreetings(2) = “world!\n”

✔ myGreetings(2) = “class!\n”

✖ myGreetings =  myStrings = Array("Big", "red",
"dad")
```

Can’t reassign to **val**

# Lists

Lists are immutable objects, cannot change the data "in place".
However, you can apply methods to Lists to generate new Lists.

```
val oneTwo = List(1, 2)
val threeFour = List(3, 4)

// using list concatenation operator (method)
val oneTwoThreeFour = oneTwo ::: threeFour
```

oneTwoThreeFour = List(1, 2, 3, 4)

**Note**: Scala is *statically typed*, so everything that goes into and out of a method is type checked,
means logic errors are likely to show up as type errors

Read this to learn more about the difference between static vs dynamic typing:
https://docs.oracle.com/cd/E57471_01/bigData.100/extensions_bdd/src/cext_transform_typing.html

# Use lists

Lists are homogenous when defined explicitly (all elements in a list have the same type.)

```
val oneTwo: List[Int] = List(1, 2)
val fruit: List[String] = List("apples","pears","plums")
val empty: List[Nothing] = List()
```

Scala list type is covariant.

If S is a subtype of T, then `List[S]` is a subtype of `List[T]`
**`empty`** has type `List[Nothing]`, so is an object of every other list type

```
val xs: List[String] = List()
```

# Lists

Heterogeneous lists are also possible in Scala

```
val oneTwo = List(1, 2, "three", 3)
val threeFour = List(3, 4)
val oneTwoThreeFour = oneTwo ::: threeFour
```

oneTwoThreeFour = List(1, 2, three, 3, 3, 4)

**oneTwo**: **List[Any]** = List(1, 2, three, 3)
**threeFour**: **List[Int]** = List(3, 4)
**oneTwoThreeFour**: **List[Any]** = List(1, 2, three, 3, 3, 4)

Read this to learn more about the difference between static vs dynamic typing:
https://docs.oracle.com/cd/E57471_01/bigData.100/extensions_bdd/src/cext_transform_typing.html

# Constructing lists

Lists are constructed recursively.
All lists are built from `Nil` and `::` ("cons")

```
val One: List[Int] = List(1, 2)
println(One)
val Two = 1 :: (2 :: Nil)
println(Two)
val fruit_group1: List[String] =
List("apples","pears","plums")
println(fruit_group1)
val fruit_group2 = "apples":: ("pears" :: ("plums" :: Nil))
println(fruit_group2)
```

**Output**

```
List(1, 2)
List(1, 2)
List(apples, pears, plums)
List(apples, pears, plums)
```

# Operations on lists

What do the following return?

```
empty.isEmpty
fruit.isEmpty
fruit.head
fruit.tail.head
fruit.last
empty.head
```

Why no built-in append method?

# Operations on lists

What do the following return?

```
empty.isEmpty          →   True
fruit.isEmpty          →   False
fruit.head             →   “apples”
fruit.tail.head        →   “pears”
fruit.last             →   “plums”
empty.head     java.util.NoSuchElementException
```

Why no built-in append method?

# Working with lists

## Sorting a list using Insertion sort : recursion

```
def isort(xs: List[Int]): List[Int] =
 if (xs.isEmpty) Nil
 else insert(xs.head, isort(xs.tail))

def insert(x: Int, xs: List[Int]); List[Int] =
  if (xs.isEmpty || x <= xs.head) x :: xs
  else xs.head :: insert(x, xs.tail)
```

# List patterns

## Insertion sort using patterns and recursion

```
val a :: b :: rest = fruit
a: String = “apples”
b: String = “pears”
rest: List[String] = List(“pears”)

def isort(xs: List[Int]): List[Int] = xs match {
  case List() => Nil
  case x :: xs1 => insert(x, isort(xs1))
}

def insert(x: Int, xs: List[Int]); List[Int] = xs match {
  case List() => List(x)
  case y :: ys => if (x <= y) x :: xs
                  else y :: insert(x, ys)
}
```

# First order methods on class List

## Concatenating lists

```
:: takes an element as left operand, list as right operand

::: takes two lists as operands

List(1,2) ::: List(3,4,5)
gives us List(1,2,3,4,5)

List() ::: List(3,4,5)
gives us List(3,4,5)

xs ::: ys ::: zs interpreted as xs ::: (ys ::: zs)
```

# Divide and conquer principle

… or patterns and recursion

```
def append[T](xs: List[T], ys: List[T]): List[T] =
  xs match {
    case List() => //??
    case x::xs1 => //??
}
```

# Divide and conquer principle

… or patterns and recursion

```
def append[T](xs: List[T], ys: List[T]): List[T] =
  xs match {
    case List() => ys
    case x::xs1 => x :: append(xs1, ys)
}
```

… what is the time complexity?
i.e. how many steps does this take?

# First order methods on lists

Scala's List class is packed with useful methods

```
List(1,2,3).length //??
List('b','b','c', 'c').last //??
List ('b','b','c', 'c').init //??

val myList = List('b','b','c', 'c').init
myList.reverse //??
```

# First order methods on lists

Scala's List class is packed with useful methods

```
List(1,2,3).length    → 3
List('b','b','c', 'c').last   → 'c'
List ('b','b','c', 'c').init  → List('b','b','c')

val myList = List('b','b','c', 'c').init
myList.reverse  → List('c','b','b')
```

# First order methods on lists

```
xs.reverse.init equals xs.tail.reverse
xs.reverse.tail equals xs.init.reverse
xs.reverse,head equals xs.last
xs.reverse.last equals xs.head

//reverse implemented using (:::)

def rev[T](xs:List[T]): List[T] = xs match {
  case List() => xs
  case x :: xs1 => rev(xs1) ::: List(x)
}
```

what is the time complexity here?

# prefixes and suffixes of lists

```
val myList = List(1,2,3,4,5)

myList take 2
myList drop 2
myList splitAt 2

myList(2)
```

# prefixes and suffixes of lists

```
val myList = List(1,2,3,4,5)

myList take 2 → List(1,2)
myList drop 2 → List(3,4,5)
myList splitAt 2 → (List(1,2), List(3,4))

myList(2) → 3  // not much used, not efficient
```

# zipping lists

`zip` operation takes two lists and forms a list of pairs

```
val myList = List(1,2,3,4,5)

myList.indices zip myList
List((0,1),(1,2),(2,3),(3,4),(4,5))

val yourlist = List('a','b','c')

myList zip yourList
List((1,'a'), (2,'b'), (3,'c')

yourList.zipWithIndex
List[(Char,Int)] = List((a,0), (b,1), (c,2))
```

# Displaying lists

```
val myList = List(1,2,3,4,5)

//toString returns canonical string representation
myList.toString

// mkString has 4 operands:
 the list to be displayed,
 a prefix string ‘pre’
 a separator string ‘sep’
 a postfix string ‘post’

myList mkString (“[”, “,”, “]”)
[1,2,3,4,5]
```

# Displaying lists

```
val myList = List(1,2,3,4,5)

// mkString variant taking only separator string
myList mkString sep equals myList mkString (“”, sep, “”)

myList mkString “” → “12345”

// mkString variant taking no arguments
myList mkString equals myList mkString “”

myList mkString → “12345”
```

# Higher-order methods on class List

```
val myList = List(1,2,3,4,5)
val yourList = List("the", "quick", "brown", "fox")

// mapping over lists
myList map (_ + 1)   → List(2,3,4,5,6)
yourList map (_.length) → List(3,5,5,3)

// filtering lists
myList filter (_ %2 == 0) → List(2,4)
yourList filter (_.length == 3) → List("the", "fox")
```

# Higher-order methods on class List

```
val myList = List(1,2,3,4,5)
val yourList = List("the", "quick", "brown", "fox")

//predicates over lists

myList forall (_ > 0)  → True
yourList exists (_.length == 3) →True


def hasZeroRow(m: List[List[Int]]) =
  m exists (row => row forall (_ == 0))

val diag3 = (List(List(1,0,0), List(0,1,0), List(0,0,1))

hasZeroRow(diag3) → False
```