Functional programming

Functional programming

- In functional programming, **functions** are the core building blocks
- In "pure" functional programming, functions are like mathematical functions
- Mathematical functions have no side effects
  - f(x) = 2x + 3
  - y = sin(x)

Functional programming

- Pure functions, in functional programming,
  - Are characterized by a set of parameters (optional)
  - Perform calculations using only the input parameters
  - Return a value (eventually a complex value)
  - Always return the same value for the same input

3

Functional programming

- Pure functions, in functional programming
  - Do not use or affect any data outside the function (they have no side effects)
    - Do not use global variables
    - Do not change the values of its parameters
  - Are not affected by any data outside the function
- Only immutable objects should be used

4

Functional programming

- If only pure functions are used
  - The code is more stable because pure functions are stateless and orthogonal to external (shared) data
    - Usually easier to find errors
  - Parallelization and concurrency is easier (no synchronization)

5

In practice

- Writing code by using only pure functions is complex and it is not always possible
- In practice,
  - The majority of the functions, in Scala functional programming, are pure functions
  - However, also some non-pure functions are used

6

Defining functions in scala

Defining functions

- Input-less function
  - def <identifier> = <expression>
- Example

  def hi = "hi"
- hi is a function returning the string "hi"

  val message: String = hi

Defining functions

- Input-less function with an explicit returned data type
  - def <identifier>: <type> = <expression>
  - Or
  - def <identifier>(): <type> = <expression>

Defining functions: Example

```
def hi:String = "hi"
val message: String = hi
-------------------------------------------------
def hi():String = "hi"
val message: String = hi
val message2: String = hi() /* The function can be
invoked by using empty parentheses in this case */
```

Defining functions

- Functions with input parameters
  - def <identifier>(<identifier>: <type>[, ...]): <type> = <expression>

Defining functions: Example

```
def multiplier(x: Int, y: Int): Int = { x * y }
/* multiplier returns x*y  */
val v1: Int=10
val v2: Int=20
val m = multiplier(v1, v2)
```

Defining functions: Example #2

```
def safeTrim(s: String): String = {
    if (s == null) null
    else s.trim()
}
val s1: String = null
val s2: String = "test     "
val s1trim = safeTrim(s1)
val s2trim = safeTrim(s2)
```

13

Procedures

- A **procedure** is a function that doesn't have a return value
  - Any function that ends with a statement, such as a println() call, is also a procedure
- If you have a function without an explicit return type that ends with a statement, the Scala compiler will infer the return type of the function to be **Unit**
  - **Unit =** no returned value

14

Procedures: Example

```
def approximate(d: Double) = println(f"Got value $d%.2f")

/* Or the equivalent definition

def approximate(d: Double): Unit = println(f"Got value $d%.2f")

*/

approximate(23.56677)
```

15

Invoking functions with an Expression Block as parameter

- An expression block can be specified as a parameter value when invoking a function
  - The value returned by the expression block is the input of the function
  - It is useful if you want to avoid the use of intermediate variables or immutable variables

16

Invoking functions with an Expression Block as parameter

```
def formatEuro(amount : Double): String  = "€"+amount
/* This function returns a string concatenating the symbol €
and the value of amount */
val res=formatEuro(3.4645)
println(res)
val res2 = formatEuro { val rate = 1.32
                        0.235 + 0.7123 + rate * 5.32 }
println(res2)
```

17

Invoking functions with an Expression Block as parameter

```
def formatEuro(amount : Double): String  = "€"+amount
/* This function returns a string concatenating the symbol €
and the value of amount */
val res=formatEuro(3.4645)
println(res)
val res2 = formatEuro { val rate = 1.32
                        0.235 + 0.7123 + rate * 5.32 }
println(res2)
```

The value returned by the expression block is used as parameter of the formatEuro function

18

Recursive functions

- Scala supports recursive functions
  - Pay attention to avoid infinite loops
- Recursive functions are very popular in functional programming because they offer a way to iterate over data structures or calculations without using mutable data

19

Recursive functions: Example

```
def power(x: Int, n: Int): Long = {
        if (n >= 1)   x * power(x, n-1)
        else 1
}

println("2^3 = "+power(2,3))
```

20

Nested functions

- Sometimes you have functions that are useful/are used only inside another function
  - They are called **nested functions**
- Nested functions are visible only inside the scope of the function in which they are defined

21

Nested functions: Example

```
def maxThree(a: Int, b: Int, c: Int) = {
    def maxTwo(x: Int, y: Int) = { if (x > y) x else y }
    maxTwo(a, maxTwo(b, c))
}

println(maxThree(5, 1, 23))
```

22

Calling Functions with Named Parameters

- The convention for calling functions is that the parameters are specified in the order in which they are originally defined
- However, in Scala you can call parameters by name, making it possible to specify them out of order
- Syntax

<function name>(<parameter> = <value>
[,<parameter> = <value>])

Calling Functions with Named Parameters: Example

```
def greet(prefix: String, name: String)=prefix+" "+name
val greeting1 = greet("Ms", "Brown")
val greeting2 = greet(name = "Brown", prefix = "Mr")
println(greeting1)
println(greeting2)
```

Parameters with Default values

- In Scala, the default value of parameters can be specified in the definition of the function
- Syntax

def <identifier>(<identifier>: <type> = <value>): <type>

25

Parameters with Default values: Example

```
def greet(prefix: String = "Ms/Mr", name: String)=prefix+" "+name

val greeting1 = greet("Ms", "Brown")

val greeting2 = greet(name = "White")

println(greeting1)

println(greeting2)
```

26

Parameters with Default values: Example

```
def greet(prefix: String = "Ms/Mr", name: String)=prefix+" "+name
val greeting1 = greet("Ms", "Brown")
val greeting2 = greet(name = "White")
println(greeting1)
println(greeting2)
```

prefix is set to the default value "Ms/Mr". Since name is the second parameter, we must use the named parameter approach to invoke the function

27

Parameters with Default values: Example #2

```
def greet(name: String, prefix: String = "Ms/Mr") = prefix+" "+name
/* The order of the parameters have been inverted */
val greeting1 = greet("Brown", "Ms")
val greeting2 = greet("White")
println(greeting1)
println(greeting2)
```

28

Parameters with Default values: Example #2

```
def greet(name: String, prefix: String = "Ms/Mr") =
prefix+" "+name
/* The order of the parameters have been inverted */
val greeting1 = greet("Brown", "Ms")
val greeting2 = greet("White")
println(greeting1)
println(greeting2)
```

prefix is set to the default value "Ms/Mr". Since name is the first parameter, we do not need to use the named parameter approach to invoke the function

29

Variable number of parameters

- Scala supports **vararg** parameters
  - You can define a function with a variable number of input arguments.
  - The vararg parameter cannot be followed by a nonvararg
- The vararg parameter is implemented as a collection
  - Inside a function it can be used, for instance, in for loops

30

Variable number of parameters

- To specify that a function parameter is a vararg parameter add an asterisk symbol (*) after the parameter's type in the function definition

31

Variable number of parameters: Example

```
def sum(items: Int*): Int = {
    var total = 0
    for (i <- items) { total += i }
    total
}
println(sum(2,3,20))
println(sum(4,10))
```

32

First class functions

First class functions

- One of the core values of functional programming is that functions should be **first class**
- It means that functions are not only declared and invoked but can be used in every segment of the language as just another data type
  - **Functions are data types**

First class functions

- A first class function may, as with other data types,
  - Be stored in a container such as a value, variable, or data structure
  - Be used as a parameter to another function or used as the return value from another function

35

First class functions

- The **function type** is a simple grouping of input types and return value type arranged with an arrow indicating the direction from input types to output type
  - i.e., the "signature" of a function without the name of the function
- Syntax

  ([<type>, ...]) => <type>

36

First class functions

- The **function type** is a simple grouping of input types and return value type arranged with an arrow indicating the direction from input types to output type
  - i.e., the "signature" of a function without the name of the function
- Syntax

Data types of the input parameters

([<type>, ...]) => <type>

37

First class functions

- The **function type** is a simple grouping of input types and return value type arranged with an arrow indicating the direction from input types to output type
  - i.e., the "signature" of a function without the name of the function
- Syntax

Data type of the returned value

([<type>, ...]) => <type>

38

First class functions

- For example, the function

  def double(x: Int): Int = x * 2

- has the function type

  Int => Int

- indicating that it has a single Int parameter and returns an Int

First class functions: Example

```
def double(x: Int): Int = x * 2

/* double is a function having an Int input parameter,
returning an Int */

println(double(5))

val myDouble: (Int) => Int = double

/* myDouble is an immutable variable.
   The type of myDouble is the function type (Int) => Int
    myDouble can be invoked a "standard" function */

println(myDouble(5))
```

First class functions: Example #2

```
def max(a: Int, b: Int) = if (a > b) a else b

/* max is a function having two Int input parameters,
returning an Int */

val maximize: (Int, Int) => Int = max

/* maximize is an immutable variable.
   The type of maximize is the function type
   (Int, Int) => Int */

val maximum=maximize(50, 30)
```

41

First class functions: Example #3

```
def logStart() = "=" * 50 + "\nStarting NOW\n" + "=" * 50

/* logStart has no input parameters and returns a String */

val start: () => String = logStart

/* start is an immutable variable.
   The type of start is the function type
   () => String */

println( start() )
```

42

Higher order functions

Higher order functions

- **Higher order functions**
  - Functions that accept other functions as parameters and/or use functions as return values are known as
- One benefit of using higher-order functions is that the actual **how** of processing the data is left as an implementation detail
- A caller can specify **what** should be done and leave the higher order functions to handle the actual logic flow

44

Higher order functions

- Each higher order function specifies the logic flow
  - i.e., the function types of the functions/procedures that must be executed and the execution flow
- The application invoking a higher order function specifies which specific function must be used to implement each "function type" defined in the flow of the higher order function

45

Higher order functions: Example

```
def safeStringOp(s: String, f: String => String) = {
    if (s != null) f(s)
    else s
}
def reverser(s: String) = s.reverse
val s1= safeStringOp(null, reverser)
val s2= safeStringOp("test", reverser)
```

46

Higher order functions: Example

```
def addPrefix(s: String) = “Prefix"+s
val s3= safeStringOp(null, addPrefix)
val s4= safeStringOp("test", addPrefix)
```

47

Higher order functions: Example #2

```
def sumOp(nums: List[Int], f: Int => Int) = {
    var sum: Int = 0
    for (v <- nums) {
        sum += f(v)
    }
    sum
}
/* Higher order function.
   It applies the function f on each element of the input
   list and returns the sum of the returned values */
```

48

Higher order functions: Example #2

```
def timesTwo(x: Int) = 2*x

/* A function having one Int as input parameter and one Int as output*/

val numbers = List(1, 5, 15)

val sum1= sumOp(numbers, timesTwo)

/* Invoke the higher order function. Use the function timesTwo as second parameter */

println(sum1)
```

49

Higher order functions

- Functions with several parameters must list them in parenthesis:

```
def test(l: List[String], f: (Int, String) => Boolean)
```

50

Function literals

- A **function literal** is a function that lacks a name
  - It is assigned to a variable or to an immutable variable
- Example

```
val doubler = (x: Int) => x * 2
/* (x: Int) => x * 2 is a function literal */
val doubled = doubler(22)
println(doubled)
```

51

Function literals

- (x: Int) => x * 2 is a function literal that defines
  - An Int input argument x
  - And the function body x * 2
- Function literals
  - Can be stored in mutable and immutable function variables
  - Defined as part of a higher-order function invocation.
- You can express a function literal in any place that accepts a function type.

52

Function literals

- Function literals have many names, depending on the used language
  - **Anonymous functions**
    - This is the Scala language's formal name for function literals
  - **Lambda expressions**
    - Both C# and Java 8 use this term, derived from the original lambda calculus syntax
  - **Lambdas**
    - A shortened version of lambda expressions.

53

Function literals

- Sintax

([<identifier>: <type>, ...]) => <expression>

54

Function literals: Example

```
val greeter = (name: String) => "Hello, "+name
/* (name: String) => "Hello, "+name
is a function literal. It concatenates the sting "Hello, "
and the input parameter name. It returns the result of
the concatenation */
val hi = greeter("World")
println(hi)
```

55

Function literals and Higher order functions

- Functional literals can be used when invoking higher order functions
  - Functional literals are used to "implement" the function parameters of higher order functions

56

Function literals and Higher order functions: Example

```
def safeStringOp(s: String, f: String => String) = {
    if (s != null) f(s)
    else s
}
/* Version without function literal */
def reverser(s: String) = s.reverse
val s1= safeStringOp(null, reverser)
val s2= safeStringOp("test", reverser)
```

57

Function literals and Higher order functions: Example

```
def safeStringOp(s: String, f: String => String) = {
    if (s != null) f(s)
    else s
}
/* Version with function literal */
val s1= safeStringOp(null, (s: String) => s.reverse)
val s2= safeStringOp("test", (s: String) => s.reverse)
```

58

Function literals and Higher order functions: Example

```
def safeStringOp(s: String, f: String => String) = {
    if (s != null) f(s)
    else s
}
/* We can omit the type of s because it can be inferred
by the type of the second parameter of safeStringOp */
val s1= safeStringOp(null, s => s.reverse)
val s2= safeStringOp("test", s => s.reverse)
```

59

Function literals and Placeholder syntax

- The **Placeholder syntax** is a shortened form of function literals, replacing named parameters with wildcard operators (_)
- It can be used when
  - (a) the explicit type of the function is specified outside the literal, and
  - (b) the parameters are used no more than once

60

Function literals and Placeholder syntax: Example

```
def safeStringOp(s: String, f: String => String) = {
    if (s != null) f(s)
    else s
}
/* Placeholder syntax */
val s1= safeStringOp(null, _.reverse)
val s2= safeStringOp("test", _.reverse)
```

61

Function literals and Placeholder syntax: Example

```
def safeStringOp(s: String, f: String => String) = {
    if (s != null) f(s)
    else s
}
/* Placeholder syntax */
val s1= safeStringOp(null, _.reverse)
val s2= safeStringOp("test", _.reverse)
```

We only specify the body of the function. The input and output types are inferred by the definition of the second parameter of safeStringOp.
_ corresponds to the first argument of the function

62

Function literals and Placeholder syntax: Example

```
def safeStringOp(s: String, f: String => String) = {
    if (s != null) f(s)
    else s
}
/* Placeholder syntax */
val s1= safeStringOp(null, _.reverse)
val s2= safeStringOp("test", _.reverse)
```

With respect to the version s => s.reverse the reference to the input parameter s has been replaced with a wildcard _ representing the first parameter of the function

63

Multiple placeholders

- You can have multiple placeholders in the same "definition"
- Since each parameter can be used only one time, when using the placeholder syntax,
  - The first _ corresponds to the first parameter of the function
  - The second _ corresponds to the second parameter of the function
  - ….

64

Multiple placeholders: Example

```
def combination(x: Int, y: Int, f: (Int,Int) => Int) =
f(x,y)

/* Higher order function that applies a function f (third
parameter) on parameters x and y and returns the value
returned by f */

combination(23, 12, _ - _)

/* Invoke combination using a function f that returns
first parameter – second parameter */
```

The first placeholder is associated with the first parameter of function f

Multiple placeholders: Example

```
def combination(x: Int, y: Int, f: (Int,Int) => Int) =
f(x,y)

/* Higher order function that applies a function f (third
parameter) on parameters x and y and returns the value
returned by f */

combination(23, 12, _ - _)

/* Invoke combination using a function f that returns
first parameter – second parameter */
```

The second placeholder is associated with the second parameter of function f

Example without placeholders

```
def combination(x: Int, y: Int, f: (Int,Int) => Int) =
f(x,y)

/* Higher order function that applies a function f (third
parameter) on parameters x and y and returns the value
returned by f */

combination(23, 12, (v, w) => v - w)

/* Invoke combination using a function f that returns v
– w. In this version each parameter has a name */
```

67

Function literal blocks and Higher order functions

- In the previous examples the bodies if the defined functional literals are single expressions
- However, expression blocks can also be used

68

Function literals and Higher order functions: Example

```
def safeStringOp(s: String, f: String => String) = {
    if (s != null) f(s)
    else s
}
/* Reverse the content of the string and apply the
toUpperCase method*/
val s1= safeStringOp("test", {s => val rev=s.reverse
                                   rev.toUpperCase})
```

69