CSC 347 - Concepts of Programming Languages

Methods and Functions: Currying

Instructor: Stefan Mitsch

Learning Objectives

❓ How are methods in object-oriented programming and functions in functional programming related?

- Understand the difference between methods and functions in Scala
- Understand the difference tupled and curried definitions
- Understand partial application

Functional Programming

- We say that functions are *first-class* if they can be
  - declared within any scope,
  - passed as arguments to other functions, and
  - returned as results of functions.
- Functions `foreach`, `map`, `filter` are *higher-order functions*
  - they take a function as argument
  - Also common: return a function as the result

Paired Methods

```scala
def add1(x:Int, y:Int) = x+y
add1(11, 21)
```

```
add1: (x: Int, y: Int)Int
res1: Int = 32
```

- This is the usual style of methods that take multiple arguments
- It is a *method* that
  - Takes a pair of `Int`s
  - Returns an `Int`

Curried Methods

```
def add2(x:Int)(y:Int) = x+y
add2(11)(21)
```

```
add2: (x: Int)(y: Int)Int
res2: Int = 32
```

- This is a curried definition
- It is a *method* that
  - Takes an Int
  - Returns a method of type `(y:Int)Int` Not to be confused with the *function* `Int=>Int`
  - So together the type of the method is `add2: (x:Int)(y:Int)Int` Not to be confused with the *function* `Int=>Int=>Int`

Functions

- Scala has first-class support for both functions and methods

Method

```scala
def plus (x:Int, y:Int) = x+y
plus(1,2)
```

Function

```scala
val plus = (x:Int, y:Int) => x+y
plus(1,2)
```

Functions

```scala
val add3 = (x:Int, y:Int) => x+y
add3(11, 21)
```

```
add3: (Int, Int) => Int = $$Lambda$4576/0x00000008018d1840@6ae4d2ad
res3: Int = 32
```

- This is a *function* that
  - Takes a pair of `Int` s
  - Returns an `Int`

Curried Functions

```
val add4 = (x:Int) => (y:Int) => x+y
add4(11)(21)
```

```
add4: Int => (Int => Int) = $$Lambda$...
res4: Int = 32
```

- This is a curried definition
- It is a *function* that
  - Takes an `Int`
  - Returns a function of type `Int=>Int`

Curried Methods

```scala
def add5(x:Int) = (y:Int) => x+y
add5(11)(21)
```

```
add5: (x: Int)Int => Int
res5: Int = 32
```

- You can mix the notations
- This is a method that
  - Takes an `Int`
  - Returns a function of type `Int=>Int`

Functions vs. Methods

```scala
def add1(x:Int, y:Int) = x+y
def add2(x:Int)(y:Int) = x+y
val add1f = add1 _
val add2f = add2 _
```

```
add1: (x: Int, y: Int)Int
add2: (x: Int)(y: Int)Int
add1f: (Int, Int) => Int = $$Lambda$...
add2f: Int => (Int => Int) = $$Lambda$...
```

- Another use of wildcard operator `_`
    - *don't care* pattern
    - anonymous function expression

Partial Application

```scala
val add4 = (x:Int) => (y:Int) => x+y
def add5(x:Int) = (y:Int) => x+y

val add4p = add4(11)
val add5p = add5(11)

val r4 = add4p(21)
val r5 = add5p(21)
```

```
add4: Int => (Int => Int) = $$Lambda$
add5: (x: Int)Int => Int

add4p: Int => Int = $$Lambda$
add5p: Int => Int = $$Lambda$

r4: Int = 32
r5: Int = 32
```

Partial Application

```scala
def add1(x:Int, y:Int) = x+y
def add2(x:Int)(y:Int) = x+y
val add3 = (x:Int, y:Int) => x+y
val add4 = (x:Int) => (y:Int) => x+y
def add5(x:Int) = (y:Int) => x+y

val add1p = add1(11, _)  /* x=>add1(11, x) */
val add2p = add2(11)(_)  /* x=>add2(11)(x) */
val add3p = add3(11, _)  /* x=>add3(11, x) */
val add4p = add4(11)
val add5p = add5(11)
val fs = List(add1p, add2p, add3p, add4p, add5p)
for f <- fs yield f(21)
```

```
fs: List[Int => Int] = List($$Lambda$,$$Lambda$,$$Lambda$,$$Lambda$,$$Lambda$)
res1: List[Int] = List(32, 32, 32, 32, 32)
```

Functions and Methods

```scala
def a (x:Int) = x + 1;
val b = (x:Int) => x + 1;
val c = new Function[Int,Int] {
  def apply(x:Int) = x + 1
}
val d : PartialFunction[Any, Int]  = {
  case i: Int => i + 1
}

val fs = List(a,b,c,d)
for f <- fs yield f(4)
```

```
fs: List[Int => Int] = List($$Lambda$, $$Lambda$, <function1>, <function1>)
res1: List[Int] = List(5, 5, 5, 5)
```

- What's going on here?
- Functions vs Methods

Functions and Methods

- `def` defines a *method* with explicit parameter types
- `=>` defines a *function* with inferable parameter types
- Functions are objects with method `apply`
    - Function `e:X=>Y` gets compiled to an object

      ```
      object e:
        def apply(x:X) : Y = ...
      ```

    - Function application `e(args)` is method invocation `e.apply(args)`

Summary

- Tupled definitions: functions with multiple arguments
- Curried definitions: a family of single-argument functions
- In Scala, functions are objects with an `apply` method
- Partial application creates new functions