CSC 347 - Concepts of Programming Languages

Methods and Functions: Currying

Instructor: James Riely

Learning Objectives

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

- Identify and describe the difference between methods and functions in Scala
- Identify and describe the difference between tupled and curried definitions
- Identify and use partial function 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

```
1 def add(x:Int, y:Int) = x+y
2 add(11, 21)
```

```
1 add: (x: Int, y: Int)Int
2 res: 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`

Functions

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

Method

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

- Part of a class structure
- Can be overridden
- Has access to fields

Function

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

- Can be passed as arguments, returned, assigned to variables

Function Notation

- Using lambda notation

```
1 val add = (x:Int, y:Int) => x+y
2 add(11,21)
```

- Use underscore when parameters used exactly once

```
1 val add = (_:Int) + (_:Int)
2 add(11,21)
```

- Types may be inferred in some contexts

```
1 var add : (Int,Int)=>Int = _ + _
2 add(11,21)
```

Curried Functions

```
1 val add = (x:Int) => (y:Int) => x+y
2 add(11)(21)
```

```
1 add: Int => (Int => Int) = $$Lambda$...
2 res: Int = 32
```

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

Curried Methods

```
1 def add(x:Int) = (y:Int) => x+y
2 add(11)(21)
```

```
1 add: (x: Int)Int => Int
2 res: Int = 32
```

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

Converting Methods to Functions

```
1 def add(x:Int, y:Int) = x+y
2 val addf = add
```

```
1 add: (x: Int, y: Int)Int
2 addf: (Int, Int) => Int = $$Lambda$...
```

Partial Application

```
1 val addf = (x:Int) => (y:Int) => x+y
2 def addm(x:Int) = (y:Int) => x+y
3
4 val addfp = addf(11)
5 val addmp = addm(11)
6
7 val rf = addfp(21)
8 val rm = addmp(21)
```

```
1 addf: Int => (Int => Int) = $$Lambda$
2 addm: (x: Int)Int => Int
3
4 addfp: Int => Int = $$Lambda$
5 addmp: Int => Int = $$Lambda$
6
7 rf: Int = 32
8 rp: Int = 32
```

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

```
1 val e = new Function[X,Y] {
2   def apply(x:X) : Y = ...
3 }
```

  - 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

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)
for f <- List(add1p, add2p, add3p, add4p, add5p)
  yield f(21)
```

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