Module 1 | IT 2nd Year Class | Functional Programming | issarapong@it.kmitl.ac.th

Functions in Programming Language

06016415 Functional Programming

Outline

- Functions in Programming Languages
- Anonymous Functions

Functions in Programming Languages

Functions: the simplicity of mathematical functions in programming languages.

Defining Functions: absolute value

Java	JavaScript	Scala	Python

int abs(int x) {
    if (x > 0) return x;
    else return -x;
}
	function abs(x) {
	    if (x > 0) return x;
	    else return -x;
	}
		def abs(x: Int): Int =
		    if x > 0 then x
		    else -x
			def abs(x):
			    return x if x > 0
			    else –x

Differences

1. A function is introduced in Java without a keyword. By contrast, JavaScript uses function, and Python and Scala use def.
2. Java, JavaScript, and Python all use the keyword return to return a value. Scala do no not.
3. The body of the function is delimited by curly braces in Java and JavaScript. Python uses indentation. For a function body as simple as the absolute value, the Scala variants use nothing.
4. Types are handled differently: Java uses a “type variable” syntax, while Scala use “variable: type.” More noticeably, JavaScript and Python do not mention types at all.
5. The languages use a different syntax to test whether input x is positive.
   - Some rely on parentheses; some don’t.
   - Some include a then keyword; others don’t.
   - Python, and Scala variants use “if” as an expression, with a value.
   - The Java and JavaScript variants do not.

Composing Functions

Imperative code tends to rely heavily on sequential composition:

pseudocode

```
doOneThing(...);
doAnotherThing(...);
```

```
def abs(x: Int): Int =
  if x > 0 then x else -x
```

```
def dots(length: Int):String =
  "." * length
```

Code	Result
dots(abs(-3))	. . .

abs(-3)
dots(-3)
	3
	error

val num = -3 val num2 = abs(num) dots(num2)	. . .
(dots compose abs)(-3)	. . .
(abs andThen dots)(-3)	. . .

Extension Methods

Extension methods are a powerful mechanism that makes it possible to seemingly add methods to an existing type.

```
def shorten(str: String, maxLen: Int): String =
  if str.length > maxLen then str.substring(0, maxLen - 3) + "..."
  else str

println(shorten("Functional programming", 20))
```

```
extension (str: String)
  def short(maxLen: Int): String =
    shorten(str, maxLen)

println("Functional programming".short(20))
```

Local Functions

An important property of local functions is that they can access the arguments and local variables of their enclosing function.

```
def abs(x: Int): Int =
  def max(a: Int, b: Int): Int = if a > b then a else b
  max(x, -x)
println(abs(-3))
```

```
def abs2(x: Int): Int =
  def maxX(a: Int): Int = if a > x then a else x
  maxX(-x)
println(abs2(-3))
```

Repeated Arguments

Many programming languages support a notion of repeated or variable-length arguments.

varargs

```
def average(first: Double, others: Double*) =
  (first + others.sum) / (1 + others.size)

average(1.0, 2.3, 4.1)
average(10.0, 20.0)
average(10.0)
```

* in the signature indicates that the argument **others** can appear 0 or more times.

Optional Arguments

Optional arguments allow programmers to specify a default value for an argument, making this argument optional when you apply the function.

```
def formatMessage(msg: String,
                  user: String = "",
                  withNewline: Boolean = true): String =
  val sb = StringBuilder()
  if user.nonEmpty then sb.append(user).append(": ")
  sb.append(msg)
  if withNewline then sb.append("\n")
  sb.result() result() similar to return

formatMessage("hello")
formatMessage("hello", "Joe")
formatMessage("hello", "Joe", false)
```

Named Arguments

With explicit names, arguments can be reordered arbitrarily.

```
formatMessage("hello", false)
formatMessage("hello", withNewline = false)
```

```
formatMessage(msg = "hello", user = "Joe")
formatMessage(user = "Joe", msg = "hello")
formatMessage(user = "Joe", withNewline = false, msg = "hello")
```

```
formatMessage("Tweedledee", "Tweedledum") // which is user and which is message?
formatMessage(msg = "Tweedledee", user = "Tweedledum")
```

Type Parameters / Parametric Polymorphism

Consider function first, which returns the first element of a pair:

```
// DON'T DO THIS!
def first(pair: (Any, Any)):
  Any = pair(0)
first((1, 2))                         // has type Any
first((1, 2)) + 10                    // rejected by the compiler
first(("egg", "chicken"))             // has type Any
first(("egg", "chicken")).toUpperCase // rejected by the compiler
```

```
def first[Type](pair: (Type, Type)):
  Type = pair(0)
```

```
def first[A](pair: (A, A)):
  A = pair(0)
first((1, 2))                         // has type Int
first((1, 2)) + 10                    // 11
first(("egg", "chicken"))             // has type String
first(("egg", "chicken")).toUpperCase // "EGG"
```

Anonymous Function

anonymous function = ฟังก์ชันที่ไม่มีชื่อ สร้างแบบไม่ใช้ def

```
val ints = List(1,2,3)
val ints = List.range(1, 10) //List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

```
val doubledInts = ints.map(_ * 2)//doubledInts: List[Int] = List(2, 4, 6)
val doubledInts = ints.map((i: Int) => i * 2)
val doubledInts = ints.map(i => i * 2)
//val doubledInts = for (i <- ints) yield i * 2
```

anonymous function

Anonymous Function with Filter

```
val ints = List.range(1, 10)
val x = ints.filter(_ > 5) //List[Int] = List(6, 7, 8, 9)
```

```
val x = ints.filter(_ < 5)
??
```

```
val x = ints.filter(_ % 2 == 0)
??
```

Summary

- Anonymous functions is a little snippets of code.
- map and filter can use on the List class.
- A lot of functionality with very little code can create with anonymous functions
- map and filter is a powerful way to create very expressive code.