Functional Programming

4 Lambda Expressions

4 Lambda Expressions

Lambda expressions

Closures

Higher-order functions

Lambda Expression and Function Objects

- **Lambda expressions** are **literals** for **creating function objects**

```
(x : Int) => x < 0
```

creates an **object** representing the **function** which can be **applied to an argument** of type Int and **returns a boolean result**

- **Function objects** are **first class objects** ➔ can be treated as any other object
  - can be **stored in variables** or **data structures**
  - can be **passed as parameter**
  - can be **returned from methods**

functions are code and code is data

- **Higher-order functions**
  - functions with **functions as parameter**
  - functions with **functions as return values**
  - functions which **create functions**
  - functions which **compose complex functions** from simpler function

Function Objects

Function parameters

```
def foreach[A] (as: List [A], action: A => Unit) =
    for (a <- as) action(a)
```

```
foreach(names, name => println(name))
```

Function as returns

```
def makePowFn(n: Double) : Double => Double =
    (x : Double) => Math.pow(x, n)
```

```
val cube = makePowFn(3.0)
val sqrt = makePowFn(0.5)
```

Function composition

```
def compose[A, B, C] (f : A => B, g : B => C) : A => C =
    (x : A) => g(f(x))
```

```
val sqrtOfCube = compose(cube, sqrt)
val y = sqrtOfCube(2.0)
```

Lambda Expressions and Function Types

Lambda expressions

```
(x : Int) => x < 0
```

```
(x : Int, y : Int ) => x + y
```

Function types

```
Int => Bool
```

```
(Int, Int) => Int
```

Function types defined as traits **`FunctionN`** with ***N*** from 0 to 22

```
(T1, T2, ..., TN) => R
```

```
trait FunctionN[-T1, -T2, ..., -TN, +R]
```

contra-variant

co-variant

```
Function0[+R]
Function1[-T1,+R]
...
Function22[-T1,-T2,...,-T22,+R]
```

for example **`Function1`**

```
trait Function1[-T1, +R] :
  def apply(t: T1) : R
  ...
```

Lambda Short Forms

Scala allows

- implicitly and explicitly typed parameters
- expression and statement body

```
(x: String)  =>  x + 1
```

explicitly typed / expression body

```
x  =>  x + 1
```

implicitly typed / expression body

```
(x, y)  =>  x + y
```

```
s  =>  {
  println(s)
  return s + 1
}
```

implicitly typed / statement body

Placeholder Syntax for Function Literals

Underscore as placeholder for lambda parameters

Explicit lambda form

```
(x : Int) => x < 0
```

short form with placeholder

```
_ < 0
```

Example:

```
val numbers = List(1, 2, 3, -2)
```

```
val negatives = numbers.filter((x : Int) => x > 0) .map((y : Int) => Math.sqrt(y))
```

Explicit lambda form

```
val negatives = numbers.filter(_ > 0) .map(Math.sqrt(_))
```

with placeholder

4 Lambda Expressions

Lambda expressions

Closures

Higher-order functions

Closures

Definition

A **closure** is a **value** storing a **function** together **with an environment**,
i.e., it contains a **mapping** associating each **free variable** of the function with the
value or **storage location** to which the variable was bound when the closure was created.

main differences in solutions of closures

Two solutions:

Haskell, Java

Solution 1: Function objects contain **values** of free variable

Solution 2: Function objects have access to **storage locations** of free variable

Scala, C#, Kotlin…

Closures in Scala

Scala implements solution 2: Capturing variables

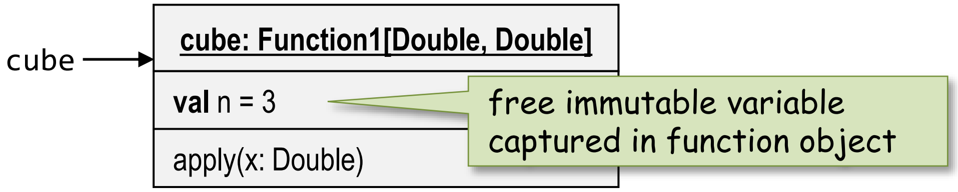
- **free local variables** are captured **in function objects**

```scala
def makePowFn(n: Double) : (Double) => Double =
    (x : Double) => Math.pow(x, n)
```

```scala
val cube = makePowFn(3)
val r = cube(4)
```

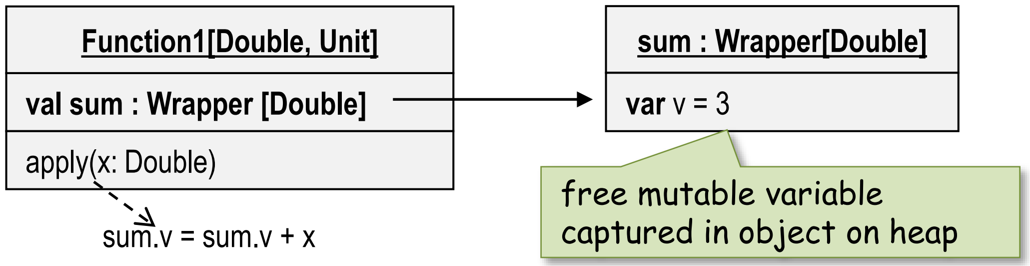
Two cases

- **Pure closures:**
  - **free variables** is **immutable**
  - ➔ **free variable becomes variable in function object**



- **Impure closures**
  - **free variable** is **mutable**
  - ➔ **free variable captured on heap**

```scala
var sum = 0.0
list.foreach(x => sum = sum + x)
```



Closures and Side Effects

- Comparison: Closures in Scala and Java

Scala

```
def sum(lst : List[Int]) : Int = {
    var sum = 0
    lst.forEach(x => {
        sum = sum + x
    })
    sum
}
```

can change free variable **sum in closure**
➔ **sum** is automatically saved on heap

Java

does not compile

```
static int sum(List<Integer> lst){
    int sum = 0;
    lst.stream().forEach(x -> {
        sum = sum + x;
    });
    return sum;
}
```

CANNOT change free variable **sum in closure**

```
static int sum(List<Integer> lst) {

    final int[] sum = new int[1];
    lst.stream().forEach(x -> {
        sum[0] = sum[0] + x;
    });
    return sum[0];
}
```

must explicitly wrap **sum** into object
➔ value of sum is final but field **sum[0]** in heap object can be change

4 Lambda Expressions

Lambda expressions

Closures

Higher-order functions

Higher-order Functions (HOF)

Scala HOF

- Declaration-site variance defined in function types

contra-variant

co-variant

```
(T1) => R   =   trait Function1[-T1, +R] {
                  def apply(t: T1) : R
                }
```

T1 for input

R for return

- No use-site variance needed

Function1[-X, +V]

```
def applyFnToX[X, V](x: X, fn: X => V ) : V = fn.apply(x)
```

```
val personNameFn : (Person) => String = (person: Person) => person.name
```

```
val student : Student = new Student("Fritz")
```

```
val info : AnyRef = applyFnToX(student, personNameFn)
```

Type inference:
X : Student
V : AnyRef
fn : Function1[-Student, +AnyRef]
personNameFn : Function1[Person, String]
➔ personNameFn compatible with fn
because Person supertype of Student
and String subtype of AnyRef