Functional Programming

5 Functional Exception Handling

Motivation

Dealing with operations which can fail

- Example: Chain of access operations in maps

```
val ssnToPerson : Map[Long, Person] = …
val pers2StdId : Map[Person, Long] = …
val stdId2Course : Map[Long, Course] = ..
val course2Room : Map[Course, Room] = ..
```

- With return values of null, a cascade of null checks is required

```
val person = ssnToPerson.apply(123424122000L)
if (person != null) {
  val id = pers2StdId.apply(person)
  if (id != 0) {
    val course = stdId2Course.apply(id)
    if (course != null) {
      val room = course2Room.apply(course)
      if (room != null) {
        println(room)
      } else {
        println("No room")
      }
    } else {
      println("No course")
    } …
```

null, if fails

Testing **null**

if cascade

Motivation

Dealing with operations which can fail

- Example: Chain of access operations in maps

```
val ssnToPerson : Map[Long, Person] = …
val pers2StdId : Map[Person, Long] = …
val stdId2Course : Map[Long, Course] = ..
val course2Room : Map[Course, Room] = ..
```

- throw exceptions

```
def lookup[K, V](map: Map[K, V], k: K) : V = {
  val v = map.apply(k)
  if (v != null) v
  else throw new NoSuchElementException()
}
```

```
val person = lookup(ssnToPerson, 123424122000L)
val id = lookup(pers2StdId, person)
val course = lookup(stdId2Course, id)
val room = lookup(course2Room, course)
println(room)
```

may throw exceptions

Scala: Option[+T]

Option with variants

- **Some** with value
- **None** for no value

```
sealed abstract class Option[+A] extends IterableOnce[A]  with Product with Serializable {
  final def isEmpty: Boolean = this eq None
  final def isDefined: Boolean = !isEmpty
  def get: A
  final def getOrElse[B >: A](default: => B): B = if (isEmpty) default else this.get
  ...
}
```

get abstract

```
final case class Some[+A](value: A) extends Option[A] {
  def get: A = value
}
```

get returing value

```
case object None extends Option[Nothing] {
  def get: Nothing = throw new NoSuchElementException("None.get")
}
```

get throwing exception

Scala: Option[+T]

Option methods

```scala
sealed abstract class Option[+A] extends IterableOnce[A]  with Product with Serializable {

  final def isEmpty: Boolean = this eq None
  final def isDefined: Boolean = !isEmpty

  def get: A

  final def getOrElse[B >: A](default: => B): B = if (isEmpty) default else this.get
  final def orElse[B >: A](alternative: => Option[B]): Option[B] = if (isEmpty) alternative else this

  def flatMap[B](f: A => Option[B]): Option[B] =
      if (isEmpty) None else f(this.get)

  def map[B](f: A => B): Option[B] =
      if (isEmpty) None else Some(f(this.get))

  def filter(p: A => Boolean): Option[A] =
      if (isEmpty || p(this.get)) this else None
  ...
}
```

SCALA: OPTION[+T]

See case study: Functional exception handling

```
trait Map[K, +V]
  def get(key: K): Option[V]
```

```
val optX : Option[Int] = bds.get("x")
```

```
if (optX.isDefined) {
  println(s"x = ${optX.get}")
} else {
  println("no value for x")
}
```

```
println(s"x = ${optX.getOrElse(-1)}")
```

```
val xOptOrElse = optX.orElse {
  print("Input value for x: ")
  option {
    scn.nextInt
  }
}
```

```
optX match {
  case Some(0)             => println("x is zero")
  case Some(x) if x < 0 => println("x is negative")
  case Some(x)             => println("x is positive")
  case None                => println("x has no value")
}
```

```
val optR2 =
  bds.get("x").flatMap {x =>
    bds.get("z").flatMap {z =>
      option {
        x / z
      }
    }
  }
```

```
val optR1 : Option[Int] =
  for (
    x <- bds.get("x");
    y <- bds.get("y");
    r <- option (x / y)
  ) yield r
```

Scala: Option[+T]

Building chains with flatMap

- Example: Chain of access operations in maps

```
val ssnToPerson : Map[Long, Person] = …
val pers2StdId : Map[Person, Long] = …
val stdId2Course : Map[Long, Course] = ..
val course2Room : Map[Course, Room] = ..
```

- Building chain of operations which may fail

```
trait Map[K, +V]
  def get(key: K): Option[V]
```

```
val optRoom : Option[Room] =
  ssnToPerson.get(123424122000L)
    .flatMap(person => pers2StdId.get(person)
      .flatMap(id => stdId2Course.get(id)
        .flatMap(course => course2Room.get(course))))

optRoom match {
  case Some(room) => println(room)
  case None => println("Not found")
}
```

flatMap chain

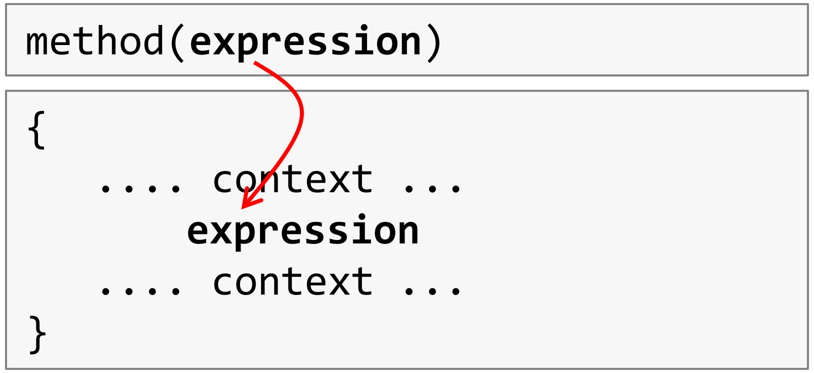
Scala Language Feature: By-name Parameters

Parameters with call-by-name semantics

```
code : => T
```

- actual argument **expressions** are passed **unevaluated**
- but **parameter** in method body is **replaced** by **expression**

```
def method[T](param : => T) = {
   .... context ...
       param
   .... context ...
}
```



```
method(expression)
```

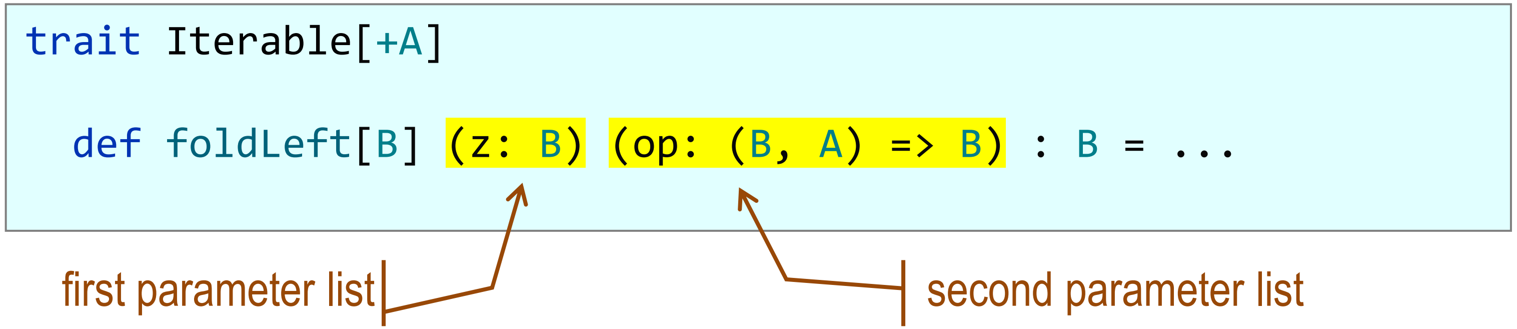
```
{
   .... context ...
       expression
   .... context ...
}
```

Note:
- by-name parameters are NO function objects (*not first-class objects*)!
- only parameter passing is call-by-name

Scala Language Feature: Multiple Parameter Lists

Scala allows multiple parameter lists

Example: foldLeft in Iterable



```scala
trait Iterable[+A]

  def foldLeft[B] (z: B) (op: (B, A) => B) : B = ...
```

```scala
val it: List[Int] = List(1, 2, 3)

val sum = it.fold (0) ((s, x) => x + s)
```

Scala Language Feature: Last Parameter List with Braces

Last parameter list can be written with braces

Example:

- foldLeft with multiple parameter lists

```
def foldLeft[B] (z: B) (op: (B, A) => B) : B = ...
```

last parameter list

- application with rounded brackets

```
list.foldLeft[B] (0) ((r, x) => r + x)
```

- application with braces

```
list.foldLeft[B] (0)  {
    (r, x) => r + x
}
```

braces!

Scala Language Feature: Last Parameter List with Braces

Last parameter list can be written with braces

- Braces also for single parameter lists

```
println { "Hallo World" }
```

- However, most often used with by-name parameter

```
def option[A](body : => A) = {
  try
    Some(body)
  catch
    case e : Exception => None
}
```

```
val optDiv: Option[Int] = option {
  x / z
}
```

```
try
   Some({
     x / z
    })
catch
   case e : Exception => None
```

Option: Usage of by-name Parameters

Example of using by-name parameters

```
sealed abstract class Option[+A] extends IterableOnce[A]  with Product with Serializable {
  ...
  final def getOrElse[B >: A](default: => B): B = if (isEmpty) default else this.get
  final def orElse[B >: A](alternative: => Option[B]): Option[B] = if (isEmpty) alternative else this
  ...
}
```

by-name parameter

```
val x = …
val y = …
val optResult : Option[Int] =
  option {
    x / y
  }
```

```
val result : Int =
  optResult.getOrElse {
    print("Input other: ")
    scanner.nextInt()
  }
```

```
if (isEmpty) {
    print("Input other: ")
    scanner.nextInt()
  } else this.get
```

by-name: replacing default by code
➔ code only executed if isEmpty

Try[+T]

Try is type analogous to Option but with

- **Success** with value or
- **Failure** with exception

```
sealed abstract class Try[+T] extends Product with Serializable {
  def isFailure: Boolean
  def isSuccess: Boolean
  def get: A
  def getOrElse[U >: T](default: => U): U
  def orElse[U >: T](default: => Try[U]): Try[U]

  def flatMap[U](f: T => Try[U]): Try[U]
  def map[U](f: T => U): Try[U]
  def filter(p: T => Boolean): Try[T]
  ...
}
```

```
final case class Success[+T](value: T) extends Try[T] {
  ...
}
```

```
final case class Failure[+T](exception: Throwable) extends Try[T] {
  ...
}
```

TRY[+T]

Constructing Try values

- Object **Try** with **apply** method

analogous to our method **option**

```
object Try {
  def apply[T](r: => T): Try[T] =
    try Success(r) catch {
      case NonFatal(e) => Failure(e)
    }
}
```

executing **r** and returning result in **Success** and catching exceptions and returning **Failure**

Example:

```
val tryDivision : Try[Int] = Try { x / y }
```

```
Try { x / y } match {
  case Success(r)  => println(s"x / y = $r")
  case Failure(e)  => println(s"x / y failed with $e")
}
```

```
val tryDiv =
  Try {
    x / y
  } orElse {
    Try {
      println("Division failed. Input value: ")
      scn.nextInt()
    }
  }
```

TRY[+T]

see more on Try in Ractive Part of course