EPFL

Recap from Weeks 1 - 6

Principles of Functional Programming

Martin Odersky

Recap: Case Classes

Case classes are Scala's preferred way to define complex data.

Example: Representing JSON (Java Script Object Notation)

```
{ "firstName" : "John",
  "lastName" : "Smith",
  "address": {
     "streetAddress": "21 2nd Street",
     "state": "NY",
     "postalCode": 10021
  },
  "phoneNumbers": [
    { "type": "home", "number": "212 555-1234" },
    { "type": "fax", "number": "646 555-4567" }
  ]
}
```

Representation of JSON with Case Classes

```
abstract class JSON
object JSON:
  case class Seq (elems: List[JSON])           extends JSON
  case class Obj (bindings: Map[String, JSON]) extends JSON
  case class Num (num: Double)                 extends JSON
  case class Str (str: String)                 extends JSON
  case class Bool(b: Boolean)                  extends JSON
  case object Null                             extends JSON
```

Representation of JSON with Enums

Case class hierarchies can be represented more concisely as enums:

```
enum JSON:
  case Seq (elems: List[JSON])
  case Obj (bindings: Map[String, JSON])
  case Num (num: Double)
  case Str (str: String)
  case Bool(b: Boolean)
  case Null
```

Example

```
val jsData = JSON.Obj(Map(
  "firstName" -> JSON.Str("John"),
  "lastName" -> JSON.Str("Smith"),
  "address" -> JSON.Obj(Map(
    "streetAddress" -> JSON.Str("21 2nd Street"),
    "state" -> JSON.Str("NY"),
    "postalCode" -> JSON.Num(10021)
  )),
  "phoneNumbers" -> JSON.Seq(List(
    JSON.Obj(Map(
      "type" -> JSON.Str("home"), "number" -> JSON.Str("212 555-1234")
    )),
    JSON.Obj(Map(
      "type" -> JSON.Str("fax"), "number" -> JSON.Str("646 555-4567")
    )) )) ))
```

Pattern Matching

Here's a method that returns the string representation of JSON data:

```
def show(json: JSON): String = json match
  case JSON.Seq(elems) =>
    elems.map(show).mkString("[", ", ", "]")
  case JSON.Obj(bindings) =>
    val assocs = bindings.map(
      (key, value) => s"${inQuotes(key)}: ${show(value)}")
    assocs.mkString("{", ",\n ", "}")
  case JSON.Num(num) => num.toString
  case JSON.Str(str) => inQuotes(str)
  case JSON.Bool(b)  => b.toString
  case JSON.Null     => "null"

def inQuotes(str: String): String = "\"" + str + "\""
```

Recap: Collections

Scala has a rich hierarchy of collection classes.

Recap: Collection Methods

All collection types share a common set of general methods.

Core methods:

```
map
flatMap
filter
```

and also

```
foldLeft
foldRight
```

Idealized Implementation of `map` on Lists

```
extension [T](xs: List[T])
  def map[U](f: T => U): List[U] = xs match
    case x :: xs1 => f(x) :: xs1.map(f)
    case Nil => Nil
```

Idealized Implementation of `flatMap` on Lists

```
extension [T](xs: List[T])
  def flatMap[U](f: T => List[U]): List[U] = xs match
    case x :: xs1 => f(x) ++ xs1.flatMap(f)
    case Nil => Nil
```

Idealized Implementation of `filter` on Lists

```
extension [T](xs: List[T])
  def filter(p: T => Boolean): List[T] = xs match {
    case x :: xs1 =>
      if p(x) then x :: xs1.filter(p) else xs1.filter(p)
    case Nil => Nil
```

Idealized Implementation of `filter` on Lists

```
extension [T](xs: List[T])
  def filter(p: T => Boolean): List[T] = xs match {
    case x :: xs1 =>
      if p(x) then x :: xs1.filter(p) else xs1.filter(p)
    case Nil => Nil
```

In practice, the implementation and type of these methods are different in order to

- make them apply to arbitrary collections, not just lists,
- make them tail-recursive on lists.

For-Expressions

Simplify combinations of core methods `map`, `flatMap`, `filter`.

Instead of:

```
(1 until n)(i =>
   (1 until i) filter (j => isPrime(i + j)) map
     (j => (i, j)))
```

one can write:

```
for
  i <- 1 until n
  j <- 1 until i
  if isPrime(i + j)
yield (i, j)
```

For-expressions and Pattern Matching

The left-hand side of a generator may also be a pattern:

```
def bindings(x: JSON): List[(String, JSON)] = x match
  case JSON.Obj(bindings) => bindings.toList
  case _ => Nil

for
  case ("phoneNumbers", JSON.Seq(numberInfos)) <- bindings(jsData)
  numberInfo <- numberInfos
  case ("number", JSON.Str(number)) <- bindings(numberInfo)
  if number.startsWith("212")
yield
  number
```

If the pattern starts with `case`, the sequence is filtered so that only elements matching the pattern are retained.

Queries with For

Principles of Functional Programming

Queries with `for`

The for notation is essentially equivalent to the common operations of query languages for databases.

Example: Suppose that we have a database `books`, represented as a list of books.

```
case class Book(title: String, authors: List[String])
```

A Mini-Database

```
val books: List[Book] = List(
  Book(title   = "Structure and Interpretation of Computer Programs",
       authors = List("Abelson, Harald", "Sussman, Gerald J.")),
  Book(title   = "Introduction to Functional Programming",
       authors = List("Bird, Richard", "Wadler, Phil")),
  Book(title   = "Effective Java",
       authors = List("Bloch, Joshua")),
  Book(title   = "Java Puzzlers",
       authors = List("Bloch, Joshua", "Gafter, Neal")),
  Book(title   = "Programming in Scala",
       authors = List("Odersky, Martin", "Spoon, Lex", "Venners, Bill")))
```

Some Queries

To find the titles of books whose author's name is "Bird":

```
for
  b <- books
  a <- b.authors
  if a.startsWith("Bird,")
yield b.title
```

To find all the books which have the word "Program'' in the title:

```
for b <- books if b.title.indexOf("Program") >= 0
yield b.title
```

Another Query

To find the names of all authors who have written at least two books present in the database.

```
for
  b1 <- books
  b2 <- books
  if b1 != b2
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
yield a1
```

Another Query

To find the names of all authors who have written at least two books present in the database.

```
for
  b1 <- books
  b2 <- books
  if b1 != b2
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
yield a1
```

Why do solutions show up twice?

How can we avoid this?

Modified Query

To find the names of all authors who have written at least two books present in the database.

```
for
  b1 <- books
  b2 <- books
  if b1.title < b2.title
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
yield a1
```

Problem

What happens if an author has published three books?

- The author is printed once
- The author is printed twice
- The author is printed three times
- The author is not printed at all

Problem

What happens if an author has published three books?

- O The author is printed once
- O The author is printed twice
- X The author is printed three times
- O The author is not printed at all

Modified Query (2)

Solution: We must remove duplicate authors who are in the results list twice.

This is achieved using the `distinct` method on sequences:

```
val repeated =
  for
    b1 <- books
    b2 <- books
    if b1.title < b2.title
    a1 <- b1.authors
    a2 <- b2.authors
    if a1 == a2
  yield a1
repeated.distinct
```

Modified Query

Better alternative: Compute with sets instead of sequences:

```
val bookSet = books.toSet
for
  b1 <- bookSet
  b2 <- bookSet
  if b1 != b2
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
yield a1
```

Translation of For

Principles of Functional Programming

For-Expressions and Higher-Order Functions

The syntax of `for` is closely related to the higher-order functions `map`, `flatMap` and `filter`.

First of all, these functions can all be defined in terms of `for`:

```
def mapFun[T, U](xs: List[T], f: T => U): List[U] =
  for x <- xs yield f(x)

def flatMap[T, U](xs: List[T], f: T => Iterable[U]): List[U] =
  for x <- xs; y <- f(x) yield y

def filter[T](xs: List[T], p: T => Boolean): List[T] =
  for x <- xs if p(x) yield x
```

Translation of For (1)

In reality, the Scala compiler expresses for-expressions in terms of `map`, `flatMap` and a lazy variant of `filter`.

Here is the translation scheme used by the compiler (we limit ourselves here to simple variables in generators)

1. A simple for-expression

```
for x <- e1 yield e2
```

is translated to

```
e1.map(x => e2)
```

Translation of For (2)

2. A for-expression

```
for x <- e1 if f; s yield e2
```

where `f` is a filter and `s` is a (potentially empty) sequence of generators and filters, is translated to

```
for x <- e1.withFilter(x => f); s yield e2
```

(and the translation continues with the new expression)

You can think of `withFilter` as a variant of `filter` that does not produce an intermediate list, but instead applies the following `map` or `flatMap` function application only to those elements that passed the test.

Translation of For (3)

3. A for-expression

```
for x <- e1; y <- e2; s yield e3
```

where `s` is a (potentially empty) sequence of generators and filters, is translated into

```
e1.flatMap(x => for y <- e2; s yield e3)
```

(and the translation continues with the new expression)

Example

Take the for-expression that computed pairs whose sum is prime:

```
for
  i <- 1 until n
  j <- 1 until i
  if isPrime(i + j)
yield (i, j)
```

Applying the translation scheme to this expression gives:

```
(1 until n).flatMap(i =>
  (1 until i)
    .withFilter(j => isPrime(i+j))
    .map(j => (i, j)))
```

This is almost exactly the expression which we came up with first!

Exercise

Translate

```
for b <- books; a <- b.authors if a.startsWith("Bird")
yield b.title
```

into higher-order functions.

Exercise

```
for b <- books; a <- b.authors if a.startsWith("Bird")
yield b.title
```

The expression above expands to which of the following two expressions?

0
```
books.flatMap(b =>
  b.authors.withFilter(a =>
    a.startsWith("Bird")).map(a => b.title))
```

0
```
books.map(b =>
  b.authors.flatMap(a =>
    if a.startsWith("Bird") then b.title))
```

Generalization of `for`

Interestingly, the translation of for is not limited to lists or sequences, or even collections;

It is based solely on the presence of the methods `map`, `flatMap` and `withFilter`.

This lets you use the for syntax for your own types as well – you must only define `map`, `flatMap` and `withFilter` for these types.

There are many types for which this is useful: arrays, iterators, databases, optional values, parsers, etc.

For and Databases

For example, `books` might not be a list, but a database stored on some server.

As long as the client interface to the database defines the methods `map`, `flatMap` and `withFilter`, we can use the `for` syntax for querying the database.

This is the basis of data base connection frameworks such as *Slick* or *Quill*, as well as big data platforms such as *Spark*.

Functional Random Generators

Principles of Functional Programming

Martin Odersky

Other Uses of For-Expressions

Question: Are for-expressions tied to collection-like things such as lists, sets, or databases?

Other Uses of For-Expressions

Question: Are for-expressions tied to collection-like things such as lists, sets, or databases?

Answer: No! All that is required is some interpretation of `map`, `flatMap` and `withFilter`.

There are many domains outside collections that afford such an interpretation.

Example: random value generators.

Random Values

You know about random numbers:

```
val rand = java.util.Random()
rand.nextInt()
```

Question: What is a systematic way to get random values for other domains, such as

- booleans, strings, pairs and tuples, lists, sets, trees

?

Generators

Let's define a trait `Generator[T]` that generates random values of type `T`:

```
trait Generator[+T]:
  def generate(): T
```

Some instances:

```
val integers = new Generator[Int]:
  val rand = java.util.Random()
  def generate() = rand.nextInt()
```

Generators

Let's define a trait `Generator[T]` that generates random values of type `T`:

```
trait Generator[+T]:
  def generate(): T
```

Some instances:

```
val booleans = new Generator[Boolean]:
  def generate() = integers.generate() > 0
```

Generators

Let's define a trait `Generator[T]` that generates random values of type `T`:

```
trait Generator[+T]:
  def generate(): T
```

Some instances:

```
val pairs = new Generator[(Int, Int)]:
  def generate() = (integers.generate(), integers.generate())
```

Streamlining It

Can we avoid the `new Generator ...` boilerplate?

Ideally, we would like to write:

```
val booleans = for x <- integers yield x > 0

def pairs[T, U](t: Generator[T], u: Generator[U]) =
  for x <- t; y <- u yield (x, y)
```

What does this expand to?

Streamlining It

Can we avoid the `new Generator ...` boilerplate?

Ideally, we would like to write:

```
val booleans = integers.map(x => x > 0)

def pairs[T, U](t: Generator[T], u: Generator[U]) =
  t.flatMap(x => u.map(y => (x, y)))
```

Need `map` and `flatMap` for that!

Generator with `map` and `flatMap`

Here's a more convenient version of `Generator`:

```
trait Generator[+T]:
  def generate(): T

extension [T, S](g: Generator[T])
  def map(f: T => S) = new Generator[S]:
    def generate() = f(g.generate())
```

Generator with `map` and `flatMap`

Here's a more convenient version of `Generator`:

```
trait Generator[+T]:
  def generate(): T

extension [T, S](g: Generator[T])
  def map(f: T => S) = new Generator[S]:
    def generate() = f(g.generate())

  def flatMap(f: T => Generator[S]) = new Generator[S]:
    def generate() = f(g.generate()).generate()
```

Generator with `map` and `flatMap` (2)

We can also implement `map` and `flatMap` as methods of class `Generator`:

```
trait Generator[+T]:
  def generate(): T

  def map[S](f: T => S) = new Generator[S]:
    def generate() = f(Generator.this.generate())
  def flatMap[S](f: T => Generator[S]) = new Generator[S]:
    def generate() = f(Generator.this.generate()).generate()
```

Note the use of `Generator.this` to the refer to the `this` of the "outer" object of class `Generator`.

The booleans Generator

What does this definition resolve to?

```
val booleans = for x <- integers yield x > 0
```

The booleans Generator

What does this definition resolve to?

```
val booleans = for x <- integers yield x > 0
```

```
val booleans = integers.map(x => x > 0)
```

The booleans Generator

What does this definition resolve to?

```
val booleans = for x <- integers yield x > 0
```

```
val booleans = integers.map(x => x > 0)
```

```
val booleans = new Generator[Boolean]:
  def generate() = ((x: Int) => x > 0)(integers.generate())
```

The booleans Generator

What does this definition resolve to?

```
val booleans = for x <- integers yield x > 0
```

```
val booleans = integers.map(x => x > 0)
```

```
val booleans = new Generator[Boolean]:
  def generate() = ((x: Int) => x > 0)(integers.generate())
```

```
val booleans = new Generator[Boolean]:
  def generate() = integers.generate() > 0
```

The pairs Generator

```
def pairs[T, U](t: Generator[T], u: Generator[U]) = t.flatMap(
  x => u.map(y => (x, y)))
```

The pairs Generator

```
def pairs[T, U](t: Generator[T], u: Generator[U]) = t.flatMap(
  x => u.map(y => (x, y)))

def pairs[T, U](t: Generator[T], u: Generator[U]) = t.flatMap(
  x => new Generator[(T, U)] { def generate() = (x, u.generate()) })
```

The pairs Generator

```
def pairs[T, U](t: Generator[T], u: Generator[U]) = t.flatMap(
  x => u.map(y => (x, y)))

def pairs[T, U](t: Generator[T], u: Generator[U]) = t.flatMap(
  x => new Generator[(T, U)] { def generate() = (x, u.generate()) })

def pairs[T, U](t: Generator[T], u: Generator[U]) = new Generator[(T, U)]:
  def generate() = (new Generator[(T, U)]:
    def generate() = (t.generate(), u.generate())
  ).generate()
```

The pairs Generator

```
def pairs[T, U](t: Generator[T], u: Generator[U]) = t.flatMap(
  x => u.map(y => (x, y)))

def pairs[T, U](t: Generator[T], u: Generator[U]) = t.flatMap(
  x => new Generator[(T, U)] { def generate() = (x, u.generate()) })

def pairs[T, U](t: Generator[T], u: Generator[U]) = new Generator[(T, U)]:
  def generate() = (new Generator[(T, U)]:
    def generate() = (t.generate(), u.generate())
  ).generate()

def pairs[T, U](t: Generator[T], u: Generator[U]) = new Generator[(T, U)]:
  def generate() = (t.generate(), u.generate())
```

Generator Examples

```
def single[T](x: T): Generator[T] = new Generator[T]:
  def generate() = x

def range(lo: Int, hi: Int): Generator[Int] =
  for x <- integers yield lo + x.abs % (hi - lo)

def oneOf[T](xs: T*): Generator[T] =
  for idx <- range(0, xs.length) yield xs(idx)
```

A `List` Generator

A list is either an empty list or a non-empty list.

```
def lists: Generator[List[Int]] =
  for
    isEmpty <- booleans
    list <- if isEmpty then emptyLists else nonEmptyLists
  yield list
```

A `List` Generator

A list is either an empty list or a non-empty list.

```
def lists: Generator[List[Int]] =
  for
    isEmpty <- booleans
    list <- if isEmpty then emptyLists else nonEmptyLists
  yield list

def emptyLists = single(Nil)
```

A `List` Generator

A list is either an empty list or a non-empty list.

```
def lists: Generator[List[Int]] =
  for
    isEmpty <- booleans
    list <- if isEmpty then emptyLists else nonEmptyLists
  yield list

def emptyLists = single(Nil)

def nonEmptyLists =
  for
    head <- integers
    tail <- lists
  yield head :: tail
```

A `Tree` Generator

Can you implement a generator that creates random `Tree` objects?

```
enum Tree:
  case Inner(left: Tree, right: Tree)
  case Leaf(x: Int)
```

Application: Random Testing

You know about unit tests:

- Come up with some some test inputs to program functions and a *postcondition*.
- The postcondition is a property of the expected result.
- Verify that the program satisfies the postcondition.

Question: Can we do without the test inputs?

Yes, by generating *random test inputs*.

Random Test Function

Using generators, we can write a random test function:

```
def test[T](g: Generator[T], numTimes: Int = 100)
           (test: T => Boolean): Unit =
  for i <- 0 until numTimes do
    val value = g.generate()
    assert(test(value), s"test failed for $value")
  println(s"passed $numTimes tests")
```

Random Test Function

Example usage:

```
test(pairs(lists, lists)) {
  (xs, ys) => (xs ++ ys).length > xs.length
}
```

Question: Does the above property always hold?

- Yes
- No

Random Test Function

Example usage:

```
test(pairs(lists, lists)) {
  (xs, ys) => (xs ++ ys).length > xs.length
}
```

Question: Does the above property always hold?

O Yes

X No

ScalaCheck

Shift in viewpoint: Instead of writing tests, write *properties* that are assumed to hold.

This idea is implemented in the `ScalaCheck` tool.

```
forAll { (l1: List[Int], l2: List[Int]) =>
  l1.size + l2.size == (l1 ++ l2).size
}
```

It can be used either stand-alone or as part of ScalaTest.

Monads

Principles of Functional Programming

Martin Odersky

Monads

Data structures with `map` and `flatMap` seem to be quite common.

In fact there's a name that describes this class of a data structures together with some algebraic laws that they should have.

They are called *monads*.

What is a Monad?

A monad `M` is a parametric type M[T] with two operations, `flatMap` and `unit`, that have to satisfy some laws.

```
extension [T, U](m: M[T])
  def flatMap(f: T => M[U]): M[U]

def unit[T](x: T): M[T]
```

In the literature, `flatMap` is also called `bind`. It can be an extension method, or be defined as a regular method in the monad class `M`.

Examples of Monads

- `List` is a monad with `unit(x) = List(x)`
- `Set` is monad with `unit(x) = Set(x)`
- `Option` is a monad with `unit(x) = Some(x)`
- `Generator` is a monad with `unit(x) = single(x)`

Monads and map

`map` can be defined for every monad as a combination of `flatMap` and `unit`:

```
m.map(f)  ==  m.flatMap(x => unit(f(x)))
          ==  m.flatMap(f andThen unit)
```

Note: `andThen` is defined function composition in the standard library.

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

Monad Laws

To qualify as a monad, a type has to satisfy three laws:

Associativity:

```
m.flatMap(f).flatMap(g)  ==  m.flatMap(f(_).flatMap(g))
```

Left unit

```
unit(x).flatMap(f)  ==  f(x)
```

Right unit

```
m.flatMap(unit)  ==  m
```

Checking Monad Laws

Let's check the monad laws for Option.

Here's `flatMap` for `Option`:

```
extension [T](xo: Option[+T])
  def flatMap[U](f: T => Option[U]): Option[U] = xo match
    case Some(x) => f(x)
    case None => None
```

Checking the Left Unit Law

Need to show: `Some(x).flatMap(f) == f(x)`

```
Some(x).flatMap(f)
```

Checking the Left Unit Law

Need to show: `Some(x).flatMap(f) == f(x)`

```
    Some(x).flatMap(f)

==  Some(x) match
      case Some(x) => f(x)
      case None => None
```

Checking the Left Unit Law

Need to show: `Some(x).flatMap(f) == f(x)`

```
    Some(x).flatMap(f)

==  Some(x) match
      case Some(x) => f(x)
      case None => None

==  f(x)
```

Checking the Right Unit Law

Need to show: `opt.flatMap(Some)` == `opt`

```
opt.flatMap(Some)
```

Checking the Right Unit Law

Need to show: `opt.flatMap(Some) == opt`

```
      opt.flatMap(Some)

==    opt match
        case Some(x) => Some(x)
        case None => None
```

Checking the Right Unit Law

Need to show: `opt.flatMap(Some) == opt`

```
      opt.flatMap(Some)

==    opt match
        case Some(x) => Some(x)
        case None => None

==    opt
```

Checking the Associative Law

Need to show: `opt.flatMap(f).flatMap(g) == opt.flatMap(f(_).flatMap(g))`

```
opt.flatMap(f).flatMap(g)
```

Checking the Associative Law

Need to show: `opt.flatMap(f).flatMap(g) == opt.flatMap(f(_).flatMap(g))`

```
       opt.flatMap(f).flatMap(g)

  ==   (opt match { case Some(x) => f(x) case None => None })
            match { case Some(y) => g(y) case None => None }
```

Checking the Associative Law

Need to show: `opt.flatMap(f).flatMap(g) == opt.flatMap(f(_).flatMap(g))`

```
       opt.flatMap(f).flatMap(g)

  ==   (opt match { case Some(x) => f(x) case None => None })
            match { case Some(y) => g(y) case None => None }

  ==   opt match
         case Some(x) =>
           f(x) match { case Some(y) => g(y) case None => None }
         case None =>
           None match { case Some(y) => g(y) case None => None }
```

Checking the Associative Law (2)

```
==   opt match
       case Some(x) =>
         f(x) match { case Some(y) => g(y) case None => None }
       case None => None
```

Checking the Associative Law (2)

```
==   opt match
       case Some(x) =>
         f(x) match { case Some(y) => g(y) case None => None }
       case None => None

==   opt match
       case Some(x) => f(x).flatMap(g)
       case None => None
```

Checking the Associative Law (2)

```
==   opt match
       case Some(x) =>
         f(x) match { case Some(y) => g(y) case None => None }
       case None => None

==   opt match
       case Some(x) => f(x).flatMap(g)
       case None => None

==   opt.flatMap(x => f(x).flatMap(g))
```

Checking the Associative Law (2)

```
==   opt match
       case Some(x) =>
         f(x) match { case Some(y) => g(y) case None => None }
       case None => None

==   opt match
       case Some(x) => f(x).flatMap(g)
       case None => None

==   opt.flatMap(x => f(x).flatMap(g))

==   opt.flatMap(f(_).flatMap(g))
```

Significance of the Laws for For-Expressions

We have seen that monad-typed expressions are typically written as `for` expressions.

What is the significance of the laws with respect to this?

1. Associativity says essentially that one can "inline" nested for expressions:

```
    for
      y <- for x <- m; y <- f(x) yield y
      z <- g(y)
    yield z

==  for x <- m; y <- f(x); z <- g(y)
    yield z
```

Significance of the Laws for For-Expressions

2. Right unit says:

```
for x <- m yield x

==  m
```

3. Left unit does not have an analogue for for-expressions.

Exceptional Monads

Principles of Functional Programming

Martin Odersky

Exceptions

Exceptions in Scala are defined similarly as in Java.

An exception class is any subclass of `java.lang.Throwable`, which has itself subclasses `java.lang.Exception` and `java.lang.Error`. Values of exception classes can be thrown.

```
class BadInput(msg: String) extends Exception(msg)
throw BadInput("missing data")
```

A thrown exception terminates computation, if it is not handled with a `try/catch`.

Handling Exceptions with `try/catch`

A `try/catch` expression consists of a *body* and one or more *handlers*.
Example:

```
def validatedInput(): String =
  try getInput()
  catch
    case BadInput(msg) => println(msg); validatedInput()
    case ex: Exception => println("fatal error; aborting"); throw ex
```

`try/catch` Expressions

An exception is caught by the closest enclosing `catch` handler that matches its type.

This can be formalized with a variant of the substitution model. Roughly:

```
    try e[throw ex] catch case x: Exc => handler
-->
    [x := ex]handler
```

Here, `ex: Exc` and `e` is some arbitrary "*evaluation context*"

- that `throw ex` as next instruction to execute and
- that does not contain a more deeply nested handler that matches `ex`.

Critique of `try/catch`

Exceptions are a low-overhead way for handling abnormal conditions.

But there have also some shortcomings.

- They don't show up in the types of functions that throw them. (in Scala, in Java they do show up in `throws` clauses but that has its own set of downsides).
- They don't work in parallel computations where we want to communicate an exception from one thread to another.

So in some situations it makes sense to see an exception as a normal function result value, instead of something special.

This idea is implemented in the `scala.util.Try` type.

Handling Exceptions with the `Try` Type

`Try` resembles `Option`, but instead of `Some`/`None` there is a `Success` case with a value and a `Failure` case that contains an exception:

```
abstract class Try[+T]
case class Success[+T](x: T)       extends Try[T]
case class Failure(ex: Exception) extends Try[Nothing]
```

A primary use of `Try` is as a means of passing between threads and processes results of computations that can fail with an exception.

Creating a Try

You can wrap up an arbitrary computation in a `Try`.

```
Try(expr)     // gives Success(someValue) or Failure(someException)
```

Here's an implementation of `Try.apply`:

```
import scala.util.control.NonFatal

object Try:
  def apply[T](expr: => T): Try[T] =
    try Success(expr)
    catch case NonFatal(ex) => Failure(ex)
```

Creating a Try

You can wrap up an arbitrary computation in a `Try`.

```
Try(expr)     // gives Success(someValue) or Failure(someException)
```

Here's an implementation of `Try.apply`:

```
import scala.util.control.NonFatal

object Try:
  def apply[T](expr: => T): Try[T] =
    try Success(expr)
    catch case NonFatal(ex) => Failure(ex)
```

Here, `NonFatal` matches all exceptions that allow to continue the program.

Composing `Try`

Just like with `Option`, `Try`-valued computations can be composed in for-expressions.

```
for
  x <- computeX
  y <- computeY
yield f(x, y)
```

If `computeX` and `computeY` succeed with results `Success(x)` and `Success(y)`, this will return `Success(f(x, y))`.

If either computation fails with an exception `ex`, this will return `Failure(ex)`.

Definition of `flatMap` and `map` on `Try`

```
extension [T](xt: Try[T])
  def flatMap[U](f: T => Try[U]): Try[U] = xt match
    case Success(x) => try f(x) catch case NonFatal(ex) => Failure(ex)
    case fail: Failure => fail

  def map[U](f: T => U): Try[U] = xt match
    case Success(x) => Try(f(x))
    case fail: Failure => fail
```

So, for a `Try` value `t`,

```
t.map(f)  ==  t.flatMap(x => Try(f(x)))
          ==  t.flatMap(f andThen Try)
```

Exercise

It looks like `Try` might be a monad, with `unit` = `Try`.

Is it?

- `Yes`
- `No, the associative law fails`
- `No, the left unit law fails`
- `No, the right unit law fails`
- `No, two or more monad laws fail.`

Solution

It turns out the left unit law fails.

```
Try(expr).flatMap(f)  !=  f(expr)
```

Indeed the left-hand side will never throw a non-fatal exception whereas the right-hand side will throw any exception thrown by `expr` or `f`.

Hence, `Try` trades one monad law for another law which is more useful in this context:

> *An expression composed from 'Try', 'map', 'flatMap' will never throw a non-fatal exception.*

Call this the "bullet-proof" principle.

Conclusion

We have seen that for-expressions are useful not only for collections.

Many other types also define `map`,`flatMap`, and `withFilter` operations and with them for-expressions.

Examples: `Generator`, `Option`, `Try`.

Many of the types defining `flatMap` are monads.

(If they also define `withFilter`, they are called "monads with zero").

The three monad laws give useful guidance in the design of library APIs.