# Lecture 6 – For Comprehensions

## SWS121: Secure Programming

Jihyeok Park

PLRG

2024 Spring

# Recall

- Basic Immutable Collections
  - Lists, Options, Maps, and Sets
- Why Immutable Collections?
- Collections Hierarchy
- Sequences
  - ArraySeq
  - Vector
  - Range
  - Queue
- Sets and Maps
  - HashSet and HashMap
  - TreeSet and TreeMap

# Contents

# Contents

# Why Monads?

Why should we understand **monads**?

# Why Monads?

Why should we understand **monads**?

**Monads** are useful tools for structuring functional programs:

# Why Monads?

Why should we understand **monads**?

**Monads** are useful tools for structuring functional programs:

- **Modularity** – They allow computations to be **composed** from simpler computations and separate the combination strategy from the actual computations being performed.

# Why Monads?

Why should we understand **monads**?

**Monads** are useful tools for structuring functional programs:

- **Modularity** – They allow computations to be **composed** from simpler computations and separate the combination strategy from the actual computations being performed.
- **Flexibility** – They allow functional programs to be much **more adaptable** than equivalent programs written without monads.

# Why Monads?

Why should we understand **monads**?

**Monads** are useful tools for structuring functional programs:

- **Modularity** – They allow computations to be **composed** from simpler computations and separate the combination strategy from the actual computations being performed.
- **Flexibility** – They allow functional programs to be much **more adaptable** than equivalent programs written without monads.
- **Isolation** – They can be used to create **imperative-style** computational structures which remain **safely isolated** from the main body of the functional program.

# Monad in Scala

In Scala, a **monad** is a **container** type that wraps **values** and provides a set of **operations** to work with the value inside the container.

# Monad in Scala

PLRG

In Scala, a **monad** is a **container** type that wraps **values** and provides a set of **operations** to work with the value inside the container.

We can define a monad with three parts in Scala:

- A **type constructor** that defines the monad type.

```
List[Int]  // A `List` monad type with `Int` as the value type
```

# Monad in Scala

In Scala, a **monad** is a **container** type that wraps **values** and provides a set of **operations** to work with the value inside the container.

We can define a monad with three parts in Scala:

- A **type constructor** that defines the monad type.

```
List[Int]  // A `List` monad type with `Int` as the value type
```

- A **type converter** that embeds a value into the monad, and we can implement it as a **constructor** or a **factory method** (`apply`) in Scala.

```
List(1, 2, 3)  // Create a `List` monad with values 1, 2, and 3
```

# Monad in Scala

In Scala, a **monad** is a **container** type that wraps **values** and provides a set of **operations** to work with the value inside the container.

We can define a monad with three parts in Scala:

- A **type constructor** that defines the monad type.

```
List[Int]  // A `List` monad type with `Int` as the value type
```

- A **type converter** that embeds a value into the monad, and we can implement it as a **constructor** or a **factory method** (`apply`) in Scala.

```
List(1, 2, 3)  // Create a `List` monad with values 1, 2, and 3
```

- A **combinator** (`flatMap` method in Scala) that applies a **monadic function** to the value inside the monad and returns a new monad.

```
List(1, 2, 3).flatMap(x => List(x, -x))  // List(1,-1,2,-2,3,-3)
```

# Monad in Scala

In Scala, we can define two more methods for a monad:

# Monad in Scala

In Scala, we can define two more methods for a monad:

- A `map` method that applies a **function** to the value inside the monad and returns a new monad.

```
List(1, 2, 3).map(x => x * 2)  // List(2, 4, 6)
```

# Monad in Scala

In Scala, we can define two more methods for a monad:

- A `map` method that applies a **function** to the value inside the monad and returns a new monad.

```
List(1, 2, 3).map(x => x * 2)  // List(2, 4, 6)
```

We can implement `map` using **type converter** and **combinator**:

```
trait List[A]:
  ...
  def map[B](f: A => B): List[B] = flatMap(x => List(f(x)))
```

# Monad in Scala

In Scala, we can define two more methods for a monad:

- A `map` method that applies a **function** to the value inside the monad and returns a new monad.

```
List(1, 2, 3).map(x => x * 2)  // List(2, 4, 6)
```

We can implement `map` using **type converter** and **combinator**:

```
trait List[A]:
  ...
  def map[B](f: A => B): List[B] = flatMap(x => List(f(x)))
```

- A `withFilter` method that applies a **predicate** to the value inside the monad and returns a new monad.

```
List(1, 2, 3).withFilter(_ % 2 == 1).map(x => x)  // List(1, 3)
```

# Monad in Scala

In Scala, we can define two more methods for a monad:

- A `map` method that applies a **function** to the value inside the monad and returns a new monad.

```
List(1, 2, 3).map(x => x * 2)  // List(2, 4, 6)
```

  We can implement `map` using **type converter** and **combinator**:

```
trait List[A]:
  ...
  def map[B](f: A => B): List[B] = flatMap(x => List(f(x)))
```

- A `withFilter` method that applies a **predicate** to the value inside the monad and returns a new monad.

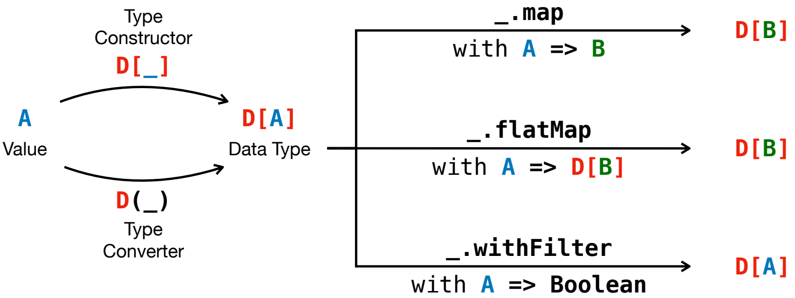
```
List(1, 2, 3).withFilter(_ % 2 == 1).map(x => x)  // List(1, 3)
```

  Or, we can simply use `filter` method:

```
List(1, 2, 3).filter(_ % 2 == 1)                  // List(1, 3)
```

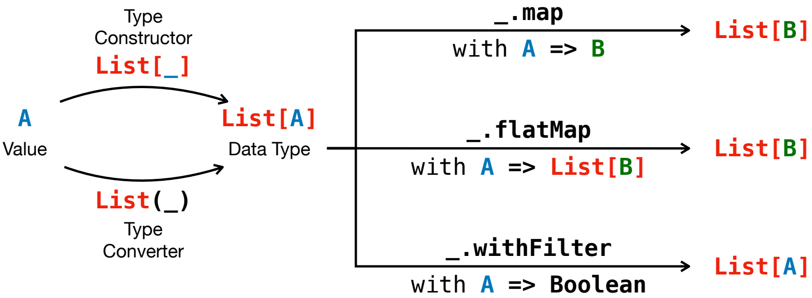
# Monad in Scala

```scala
// type constructor
trait D[A]:
  def map[B](f: A => B): D[B] = ???             // `map`
  def flatMap[B](f: A => D[B]): D[B] = ???      // `flatMap` (combinator)
  def withFilter(p: A => Boolean): D[A] = ??? // `withFilter`

object D:
  def apply[A](value: A): D[A] = ???           // type converter
```



# Monad in Scala – List Monad

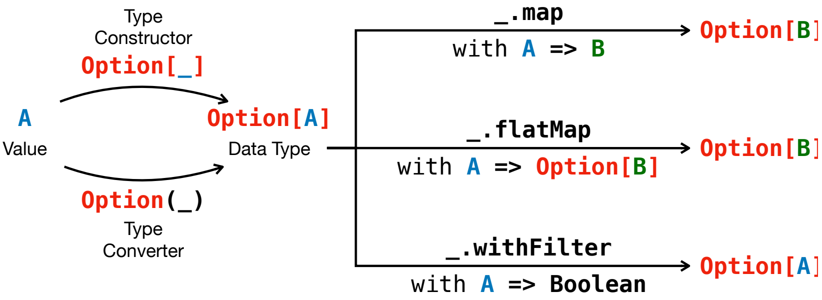
```scala
val list: List[Int] = List(1, 2)         // List(1, 2)

list.map("a" * _)                        // List("a", "aa")
list.flatMap(x => List(x%2 == 0, x<2)) // List(false, true, true, false)
list.withFilter(_ % 2 == 1)              // List(1, 3)
// In fact, we need _.map(x => x) to get List(1, 3)
// Or, use `filter` instead
list.filter(_ % 2 == 1)                  // List(1, 3)
```



# Monad in Scala – Option Monad

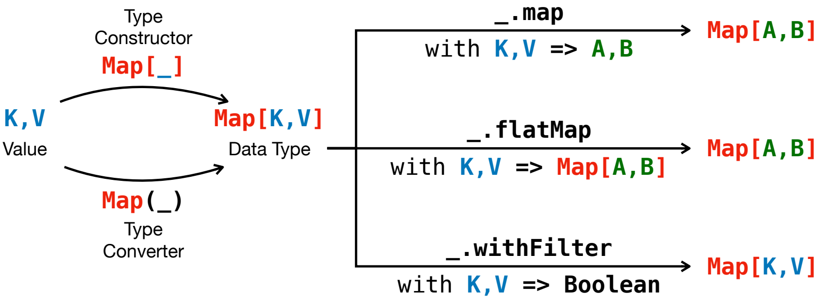
PLRG

```scala
val some: Option[Int] = Some(3)     // Some(1)
val none: Option[Int] = None        // None
some.map("a" * _)                   // Some("aaa")
none.map("a" * _)                   // None
some.flatMap(x => Some(x < 2))      // Some(false)
none.flatMap(x => Some(x < 2))      // None
some.filter(_ % 2 == 1)             // Some(1)
none.filter(_ % 2 == 1)             // None
```



# Monad in Scala – Map Monad

```scala
val map: Map[Int, String] = Map(1 -> "abc", 2 -> "d")

map.map { case (k, v) => (k, v.length) }
// Map(1 -> 3, 2 -> 1)
map.flatMap { case (k, v) => Map(k -> v, -k -> v.reverse) }
// Map(1 -> "abc", -1 -> "cba", 2 -> "d", -2 -> "d")
map.filter { case (k, v) => k % 2 == 1 }
// Map(1 -> "abc")
```



# Monad Laws

There are **three laws** that a monad must obey:

- **Left Identity**

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

- **Right Identity**:

```
m.flatMap(apply) == m
```

- **Associativity**:

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

# For Comprehensions

Scala supports **for-comprehensions** as a syntactic sugar to work with operations on **monads** in a more **imperative** way.

[1] https://docs.scala-lang.org/tour/for-comprehensions.html

# For Comprehensions

Scala supports **for-comprehensions** as a syntactic sugar to work with operations on **monads** in a more **imperative** way.

A **for-comprehension**[1] is a syntactic sugar:

```
val list = List(1, 2, 3)
for {
  x <- list if x % 2 == 1
  y <- List(x, -x)
} yield x * y
```

[1] https://docs.scala-lang.org/tour/for-comprehensions.html

# For Comprehensions

Scala supports **for-comprehensions** as a syntactic sugar to work with operations on **monads** in a more **imperative** way.

A **for-comprehension**[1] is a syntactic sugar:

```
val list = List(1, 2, 3)
for {
  x <- list if x % 2 == 1
  y <- List(x, -x)
} yield x * y
```

is equivalent to:

```
list
  .withFilter(x => x % 2 == 1)
  .flatMap(x =>
    List(x, -x)
      .map(y => x * y)
  )
```

[1] https://docs.scala-lang.org/tour/for-comprehensions.html

The **for-comprehension** syntax also supports **pattern matching**:

```
enum Shape:
  case Circle(radius: Int)
  case Rectangle(width: Int, height: Int)
import Shape.*
val shapes = List(Rectangle(2, 3), Circle(4), Rectangle(5, 6))
```

# For Comprehensions

The **for-comprehension** syntax also supports **pattern matching**:

```
enum Shape:
  case Circle(radius: Int)
  case Rectangle(width: Int, height: Int)
import Shape.*
val shapes = List(Rectangle(2, 3), Circle(4), Rectangle(5, 6))
```

```
for { // a list of areas of only rectangles in the list
  case Rectangle(width, height) <- shapes
} yield width * height                  // List(6, 30)
```

# For Comprehensions

The **for-comprehension** syntax also supports **pattern matching**:

```
enum Shape:
  case Circle(radius: Int)
  case Rectangle(width: Int, height: Int)
import Shape.*
val shapes = List(Rectangle(2, 3), Circle(4), Rectangle(5, 6))
```

```
for { // a list of areas of only rectangles in the list
  case Rectangle(width, height) <- shapes
} yield width * height                    // List(6, 30)
```

is equivalent to:

```
shapes.withFilter {
  case Rectangle(_, _) => true
  case _ => false
}.map {
  case Rectangle(width, height) => width * height
}
```

# For Comprehensions

All **immutable collections** in Scala are **monads**.

# For Comprehensions

All **immutable collections** in Scala are **monads**.

Since they share the same **Iterable** trait, we can mix them in a single **for-comprehension** and freely convert between them.

```
val list: List[(Int, String)] = for {
  x <- List(1, 2, 3)
  if x % 2 == 1
  y <- Set(x - 1, x, x + 1)
  z <- if (y % 2 == 0) Some(y) else None
} yield (x, "a" * z)
// List((1, ""), (1, "aa"), (3, "aa"), (3, "aaaa"))

// Converting a list of tuples to a map
val map: Map[Int, String] = list.toMap
// Map(1 -> "aa", 3 -> "aaaa")
```

# For Comprehensions

Most data structures in Scala are **monads**:

- All collections (subtypes of `Iterable` trait) in Scala
  - `Seq` – A sequence of elements (e.g., `List`, `Vector`, `Range`, `Queue`, etc.)
  - `Set` – A set of unique elements (e.g., `HashSet`, `TreeSet`, etc.)
  - `Map` – A map of key-value pairs (e.g., `HashMap`, `TreeMap`, etc.)
- **Functional error handling**
  - `Option` – `Some` for success, `None` for failure
  - `Try` – `Success` for success, `Failure` for failure
  - `Either` – `Left` for failure, `Right` for success
- **Concurrency**
  - `Future` – A value that will be available at some point

In addition, **Scalaz**[2] and **Cats**[3] libraries provide more functional programming abstractions.

[2] https://scalaz.github.io/
[3] https://typelevel.org/cats/

# Example 1 – Options

```
def makeInt(s: String): Option[Int] =
  try Some(Integer.parseInt(s)) catch case _: Exception => None
```

Let's define a function `addStrings` that takes three strings and returns the sum of the corresponding integers using `makeInt`.

# Example 1 – Options

PLRG

```
def makeInt(s: String): Option[Int] =
  try Some(Integer.parseInt(s)) catch case _: Exception => None
```

Let's define a function `addStrings` that takes three strings and returns the sum of the corresponding integers using `makeInt`.

Without **for-comprehension**, the implementation becomes too verbose:

```
def addStrings(s1: String, s2: String, s3: String): Option[Int] =
  makeInt(s1) match
    case Some(a) =>
      makeInt(s2) match
        case Some(b) =>
          makeInt(s3) match
            case Some(c) => Some(a + b + c)
            case None => None
        case None => None
    case None => None
```

# Example 1 – Options

PLRG

```
def makeInt(s: String): Option[Int] =
  try Some(Integer.parseInt(s)) catch case _: Exception => None
```

Let's define a function `addStrings` that takes three strings and returns the sum of the corresponding integers using `makeInt`.

With **for-comprehension**, the implementation becomes more concise:

```
def addStrings(s1: String, s2: String, s3: String): Option[Int] = for {
  a <- makeInt(s1)
  b <- makeInt(s2)
  c <- makeInt(s3)
} yield a + b + c
```

```
addStrings("1", "2", "3")      // Some(6)
addStrings("x", "2", "3")      // None
```

# Example 2 – Lists

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

Consider a simple database of books, represented as a list of Book objects:

```scala
val books: List[Book] = List(
  Book(
    "Theory of Programming Languages",
    List("John C. Reynolds"),
    1998),
  Book(
    "Types and Programming Languages",
    List("Benjamin C. Pierce"),
    2002),
  Book(
    "Automata Theory, Languages, and Computation",
    List("John E. Hopcroft", "Rajeev Motwani", "Jeffrey D. Ullman"),
    2006),
  Book(
    "Compilers: Principles, Techniques, and Tools",
    List("Alfred V. Aho", "Monica S. Lam", "Ravi Sethi", "Jeffrey D. Ullman"),
    2006),
)
```

# Example 2 – Lists

PLRG

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

**Find the titles of books whose authors has last name "Ullman"**:

# Example 2 – Lists

PLRG

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

**Find the titles of books whose authors has last name "Ullman":**

```
for {
  book <- books
  author <- book.authors
  if author.endsWith("Ullman")
} yield book.title
```

# Example 2 – Lists

PLRG

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

**Find the titles of books whose authors has last name "Ullman"**:

```
for {
  book <- books
  author <- book.authors
  if author.endsWith("Ullman")
} yield book.title
```

**Find all pairs of books written by at least one common author**:

# Example 2 – Lists

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

**Find the titles of books whose authors has last name "Ullman"**:

```
for {
  book <- books
  author <- book.authors
  if author.endsWith("Ullman")
} yield book.title
```

**Find all pairs of books written by at least one common author**:

```
for {
  book1 <- books
  book2 <- books
  if book1 != book2
  author1 <- book1.authors
  author2 <- book2.authors
  if author1 == author2
} yield (book1, book2)
```

# Example 3 – Maps

```scala
val map: Map[Int, List[Int]] = Map(
  1 -> List(3, 2, 10),
  2 -> List(4, 5),
  3 -> List(6, 7, 8, 2),
  5 -> List(9, 10),
)
val keys: Set[Int] = Set(1, 3)
```

**Find set of even values in the value lists for given keys in the map**:

# Example 3 – Maps

PLRG

```scala
val map: Map[Int, List[Int]] = Map(
  1 -> List(3, 2, 10),
  2 -> List(4, 5),
  3 -> List(6, 7, 8, 2),
  5 -> List(9, 10),
)
val keys: Set[Int] = Set(1, 3)
```

**Find set of even values in the value lists for given keys in the map**:

```scala
val list = for {
  (key, values) <- map
  if keys.contains(key)
  value <- values
  if value % 2 == 0
} yield value
// List(2, 10, 6, 8, 2)
val set = list.toSet
// Set(2, 6, 8, 10)
```

# Contents

# Tree Monad

Can we define a **tree monad**?

# Tree Monad

Can we define a **tree monad**?

Let's define a **tree monad** with 1) an integer and 2) sub-trees as children.

```
case class Tree(value: Int, children: List[Tree]):
  def map(f: Int => Int): Tree = Tree(f(value), children.map(_.map(f)))
  def flatMap(f: Int => Tree): Tree =
    val Tree(v, cs) = f(value)
    Tree(v, cs ++ children.map(_.flatMap(f)))

object Tree:
  def apply(value: Int): Tree = Tree(value, Nil)
```

# Tree Monad

Can we define a **tree monad**?

Let's define a **tree monad** with 1) an integer and 2) sub-trees as children.

```
case class Tree(value: Int, children: List[Tree]):
  def map(f: Int => Int): Tree = Tree(f(value), children.map(_.map(f)))
  def flatMap(f: Int => Tree): Tree =
    val Tree(v, cs) = f(value)
    Tree(v, cs ++ children.map(_.flatMap(f)))

object Tree:
  def apply(value: Int): Tree = Tree(value, Nil)
```

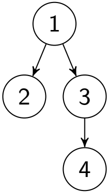
We can verify that the **tree monad** obeys the three **monad laws**:

```
1) Tree(x).flatMap(f) == f(x)                // Left Identity
2) m.flatMap(Tree.apply) == m                // Right Identity
3) m.flatMap(f).flatMap(g)
   == m.flatMap(x => f(x).flatMap(g))        // Associativity
```

# Tree Monad – Application

Let's utilize the **tree monad** to modify the values in a tree:

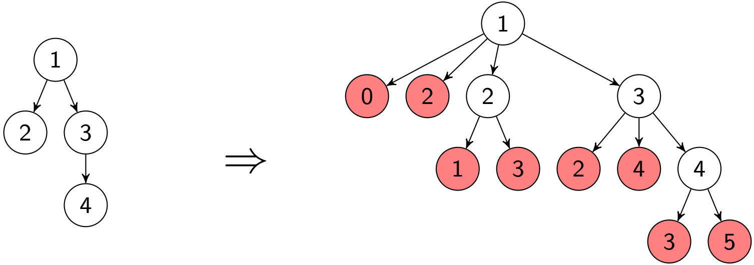
```
val tree = Tree(1, List(Tree(2), Tree(3, List(Tree(4)))))
```



# Tree Monad – Application

Let's utilize the **tree monad** to modify the values in a tree:

```
val tree = Tree(1, List(Tree(2), Tree(3, List(Tree(4)))))
```




```
for {
  x <- tree
  y <- Tree(x, List(Tree(x - 1), Tree(x + 1)))
} yield y
```

# State Monad – Motivation

A pure functional programming does **not allow mutable state**.

# State Monad – Motivation

A pure functional programming does **not allow mutable state**.

However, we often require **stateful computations**.

# State Monad – Motivation

A pure functional programming does **not allow mutable state**.

However, we often require **stateful computations**.

Then, we can mimic them by returning **updated states** along with **results**:

```scala
case class Stack(values: List[Int]):
  def push(value: Int): Stack = Stack(value :: values)
  def pop: (Stack, Option[Int]) = values match
    case Nil => (this, None)
    case x :: xs => (Stack(xs), Some(x))
```

# State Monad – Motivation

A pure functional programming does **not allow mutable state**.

However, we often require **stateful computations**.

Then, we can mimic them by returning **updated states** along with **results**:

```
case class Stack(values: List[Int]):
  def push(value: Int): Stack = Stack(value :: values)
  def pop: (Stack, Option[Int]) = values match
    case Nil => (this, None)
    case x :: xs => (Stack(xs), Some(x))
```

```
val s0 = Stack(Nil)              // s0 = Stack(List())
val s1 = s0.push(3)              // s1 = Stack(List(3))
val s2 = s1.push(7)              // s2 = Stack(List(7, 3))
val (s3, v1) = s2.pop            // s3 = Stack(List(3)),   v1 = Some(7)
val (s4, v2) = s3.pop            // s4 = Stack(List()),    v2 = Some(3)
val (s5, v3) = s4.pop            // s5 = Stack(List()),    v3 = None
val s6 = s5.push(5)              // s6 = Stack(List(5))
List(v1, v2, v3).flatten.sum     // 10
```

# State Monad – Definition

A **state monad** encapsulates a **stateful computation**, a **function** that

- **takes** the **current state** and
- **returns** 1) the **updated state** along with 2) the **computation result**.

```
case class State[S, A](compute: S => (S, A)):
  def map[B](f: A => B): State[S, B] = flatMap(x => State(f(x)))
  def flatMap[B](f: A => State[S, B]): State[S, B] = State(s => {
    val (s1, a) = compute(s)
    f(a).compute(s1)
  })
  // No `withFilter` method for `State`
object State:
  def apply[S, A](a: A): State[S, A] = State(s => (s, a))
```

# State Monad – Definition

PLRG

A **state monad** encapsulates a **stateful computation**, a **function** that

- **takes** the **current state** and
- **returns** 1) the **updated state** along with 2) the **computation result**.

```
case class State[S, A](compute: S => (S, A)):
  def map[B](f: A => B): State[S, B] = flatMap(x => State(f(x)))
  def flatMap[B](f: A => State[S, B]): State[S, B] = State(s => {
    val (s1, a) = compute(s)
    f(a).compute(s1)
  })
  // No `withFilter` method for `State`
object State:
  def apply[S, A](a: A): State[S, A] = State(s => (s, a))
```

We can verify that the **state monad** obeys the three **monad laws**:

```
1) State(x).flatMap(f) == f(x)              // Left Identity
2) m.flatMap(State.apply) == m              // Right Identity
3) m.flatMap(f).flatMap(g)
   == m.flatMap(x => f(x).flatMap(g))       // Associativity
```

# State Monad – Application

Now, add helper methods to the **stack** using the **state monad**:

```
object Stack:
  def push(v: Int): State[Stack, Unit] = State(s => (s.push(v), ()))
  def pop: State[Stack, Option[Int]] = State(_.pop)
```

# State Monad – Application

PLRG

Now, add helper methods to the **stack** using the **state monad**:

```
object Stack:
  def push(v: Int): State[Stack, Unit] = State(s => (s.push(v), ()))
  def pop: State[Stack, Option[Int]] = State(_.pop)
```

Then, we can rewrite the previous example using the **state monad**:

```
import Stack.*, State.*
val state = for {
  _ <- push(3)
  _ <- push(7)
  v1 <- pop
  v2 <- pop
  v3 <- pop
  _ <- push(5)
} yield List(v1, v2, v3).flatten.sum
state.compute(Stack(Nil))          // (Stack(List()), 10)
```

# State Monad – Application

PLRG

Now, add helper methods to the **stack** using the **state monad**:

```
object Stack:
  def push(v: Int): State[Stack, Unit] = State(s => (s.push(v), ()))
  def pop: State[Stack, Option[Int]] = State(_.pop)
```

Then, we can rewrite the previous example using the **state monad**:

```
import Stack.*, State.*
val state = for {
  _ <- push(3)
  _ <- push(7)
  v1 <- pop
  v2 <- pop
  v3 <- pop
  _ <- push(5)
} yield List(v1, v2, v3).flatten.sum
state.compute(Stack(Nil))         // (Stack(List()), 10)
```

We can **reuse** the computation with **different initial states**:

```
state.compute(Stack(List(1, 2)))  // (Stack(List(5, 2)), 11)
```

# Summary

1. Monads
    - Why Monads?
    - Monad in Scala
    - Monad Laws
    - For Comprehensions
    - Examples

2. Custom Monads
    - Tree Monad
    - State Monad

# Next Lecture

- Lazy Evaluation

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr