# Lecture 8 – Generics

## SWS121: Secure Programming

Jihyeok Park

PLRG

2024 Spring

# Recall

- Lazy Values (`lazy val`)
  - Call-By-Need Evaluation
  - Why Lazy Values?
- By-Name Parameters
  - Call-By-Need vs Call-By-Name
  - Examples
  - By-Name Parameters with Lazy Values
- Lazy Lists
  - Example: Natural Numbers
  - Example: Even Numbers
  - Example: Fibonacci Numbers
  - Example: Prime Numbers
- Views for Collections
  - Example: Find Palindromes

# Contents

# Contents

# Generic Classes

**Generic classes** take a **type parameter** within square brackets `[]`.

Most collection classes in Scala are generic.

For example, `List[T]` is generic, where `T` is the type of elements.

To use a generic class, put any **type argument** in place of `T`.

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

val strList: List[String] = List("a", "b", "c")

enum Fruit { case Apple, Orange }
import Fruit.*
val fruitList: List[Fruit] = List(Apple, Orange, Orange)
```

We need to follow the type rules when using generic classes.

```
// Type Mismatch Error: `Int` required but `String` found
val intList: List[Int] = List(1, 2, "a")
```

# Generic Classes

Let's define a simple **generic class** `Stack[T]` that can store elements of a given type `T` in a stack.

```
class Stack[T]:
  private var elements: List[T] = Nil
  def push(x: T): Unit = elements = x :: elements
  def peek: T = elements.head
  def pop: T =
    val currentTop = peek
    elements = elements.tail
    currentTop
```

```
val stack = Stack[Int]
stack.push(1)
stack.push(2)
println(stack.pop) // 2
println(stack.pop) // 1
// Type Mismatch Error: `Int` required but `String` found
stack.push("abc")
```

# Generic Classes – Algebraic Data Types (ADTs)

We can apply generics to **algebraic data types (ADTs)** as well.

Let's define an ADT `Expr[T]` for expressions with values of type `T`.

```
enum Expr[T]:
  case Val(value: T)
  case Add(left: Expr[T], right: Expr[T])
  case Mul(left: Expr[T], right: Expr[T])
```

```
import Expr.*
// 1 + (2 * 3)
val expr1: Expr[Int] = Add(Val(1), Mul(Val(2), Val(3)))
// "a" + ("b" * "c")
val expr2: Expr[String] = Add(Val("a"), Mul(Val("b"), Val("c")))

enum Binary { case Zero, One }
import Binary.*
// 0 + (1 * 0)
val expr3: Expr[Binary] = Add(Val(Zero), Mul(Val(One), Val(Zero)))
```

# Generic Classes – Subtypes

We can pass a value whose type is a **subtype** of the **type argument**.

```
sealed trait Animal { def name: String }
case class Cat(name: String) extends Animal
case class Dog(name: String) extends Animal

// `Cat` and `Dog` are subtypes of `Animal`
val animalList: List[Animal] = List(Cat("Alice"), Dog("Bob"))
```

In Scala,

- `Any` is the **supertype** of all types, and **all values** are instances of `Any`.
- `Nothing` is the **subtype** of all types, and **no instances** for `Nothing`.

```
// We can insert any value into `List[Any]`
val anyList: List[Any] = List(1, "abc", Cat("Alice"))

// We cannot insert any value into `List[Nothing]`
val nothingList: List[Nothing] = Nil
```

# Contents

# Generic Methods/Functions

We can define **generic methods** or **functions** that take **type parameters**.

```scala
def repeat[T](x: T, n: Int): List[T] =
  if (n == 0) Nil
  else x :: repeat(x, n - 1)
```

We sometimes call them **polymorphic methods** or **functions** because they can operate on values of different types.

```scala
println(repeat[Int](42, 5))                // List(42, 42, 42, 42, 42)

println(repeat[String]("abc", 3))          // List("abc", "abc", "abc")

enum Fruit { case Apple, Orange }
println(repeat[Fruit](Fruit.Apple, 2))     // List(Apple, Apple)
```

The type parameter `T` is inferred from the given arguments.

```scala
println(repeat(42, 5)) // `T` is inferred as `Int` because `42` is `Int`
```

# Contents

# Type Bounds

We can specify **type bounds** for type parameters.

- **Upper Type Bound**: `T <: U` means `T` must be a **subtype** of `U`.
- **Lower Type Bound**: `T >: L` means `T` must be a **supertype** of `L`.

```scala
sealed trait Animal { def name: String }
case class Cat(name: String) extends Animal
case class Dog(name: String) extends Animal

case class Pair[T <: Animal, U >: Animal](left: T, right: U)
```

```scala
Pair[Animal, Animal](Cat("Alice"), Dog("Bob"))
Pair[Animal, Any](Cat("Alice"), 42)
Pair[Cat, Animal](Cat("Alice"), Dog("Bob"))

// Type Mismatch Error: `Any` is not a subtype of `Animal`
Pair[Any, Animal]("abc", Dog("Bob"))

// Type Mismatch Error: `Cat` is not a supertype of `Animal`
Pair[Animal, Cat](Dog("Bob"), Cat("Alice"))
```

# Contents

# Variances

```scala
case class Box[T](value: T)

val intBox: Box[Int] = Box[Int](42)

// Type Mismatch Error: `Box[Int]` is not a subtype of `Box[Any]`
val anyBox: Box[Any] = intBox
```

However, we can do similar things with `List`.

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

// Possible: `List[Int]` is a subtype of `List[Any]`
val anyList: List[Any] = intList
```

The difference is that `List` is **covariant** to its type parameter, `Box` is not.

Let's learn about **variances** in Scala.

# Variances

**Variances** specify how the **subtyping relationship** of a class should be **inherited** by its **type parameters**.

There are three types of variances:

- **Invariance** `A[T]`:

  if $\mathtt{T} \neq \mathtt{U}$, then **no subtyping relationship** between `A[T]` and `A[U]`

- **Covariance** `A[+T]`:

  if `T <: U`, then `A[T] <: A[U]`

- **Contravariance** `A[-T]`:

  if `T <: U`, then `A[T] >: A[U]`

# Variances – Invariance

The following is an example of **invariant class** `Box[T]`.

```scala
// An invariant class `Box[T]`
case class Box[T](value: T)
```

There is **no subtyping relationship** between `Box[T]` and `Box[U]` if `T` and `U` are different types, even though `T <: U` or `U <: T`.

```scala
sealed trait Animal { def name: String }
case class Cat(name: String) extends Animal
case class Dog(name: String) extends Animal

// Type Mismatch Error: `Box[Cat]` is not a subtype of `Box[Animal]`
// even though `Cat` is a subtype of `Animal`
val animalBox: Box[Animal] = Box[Cat](Cat("Alice"))

// Type Mismatch Error: `Box[Animal]` is not a subtype of `Box[Dog]`
// even though `Dog` is a subtype of `Animal`
val dogBox: Box[Dog] = Box[Animal](Dog("Bob"))
```

# Variances – Covariance

We can make the class Box **covariant** by adding a + to the type parameter.

```
// A covariant class `Box[+T]`
case class Box[+T](value: T)
```

If T is a **subtype** of U, then Box[T] is a **subtype** of Box[U].

$$\text{if } \texttt{T <: U}, \text{ then } \texttt{Box[T] <: Box[U]}$$

```
sealed trait Animal { def name: String }
case class Cat(name: String) extends Animal
case class Dog(name: String) extends Animal

// `Box[Cat]` is a subtype of `Box[Animal]` because `Cat <: Animal`
val animalBox: Box[Animal] = Box[Cat](Cat("Alice"))

// Type Mismatch Error: `Box[Animal]` is not a subtype of `Box[Dog]`
// `Animal` is not a subtype of `Dog`
val dogBox: Box[Dog] = Box[Animal](Dog("Bob"))
```

# Variances – Covariance

The `Option` and `List` classes in Scala are **covariant**.

```
// Covariant class `Option[+T]`
val opt: Option[Animal] = Option[Cat](Cat("Alice"))

// Covariant class `List[+T]`
val list: List[Animal] = List[Dog](Dog("Bob"), Dog("Charlie"))
```

`None` and `Nil` are `Option[Nothing]` and `List[Nothing]`, respectively.

Since, `Nothing` is a **subtype** of all types, `None` and `Nil` can be assigned to `Option[T]` and `List[T]` for any type `T`.

```
val opt: Option[Animal] = None // Option[Nothing]
val list: List[Animal] = Nil // List[Nothing]
```

Note that `Set` is **invariant** in Scala.

```
// Type Mismatch Error: `Set[Int]` is not a subtype of `Set[Any]`
val set: Set[Any] = Set[Int](1, 2, 3)
```

# Variances – Contravariance

The **contravariance** is the opposite of covariance.

If `T` is a **subtype** of `U`, then `Box[T]` is a **supertype** of `Box[U]`.

if `T <: U`, then `Box[T] >: Box[U]`

The common use case of contravariance is for **function arguments**:

```
// A contravariant class `Stringifier[-T]`
trait Stringifier[-T] { def stringify(x: T): String }
```

```
sealed trait Animal { def name: String }
case class Cat(name: String) extends Animal
case class Dog(name: String) extends Animal

val animalStringifier: Stringifier[Animal] = new Stringifier[Animal]:
  def stringify(x: Animal): String = x.name

val catStringifier: Stringifier[Cat] = animalStringifier
```

It is safe to pass a `Cat` to a function that expects an `Animal`.

# Variances

Therefore, the subtyping relationship between function types is:

- **contravariant** in the **argument type** and
- **covariant** in the return type.

  if `I2 <: I1` and `O1 <: O2`, then `(I1 => O1) <: (I2 => O2)`

For example,

```
val intToInt: Int => Int = x => x + 1

// (Int => Any) <: (Nothing => Int)
// because Nothing <: Int and argument type is contravariant
val nothingToInt: Nothing => Int = intToInt

// (Int => Int) <: (Int => Any)
// becuase Int <: Any and return type is covariant
val intToAny: Int => Any = intToInt
```

# Contents

# Abstract Type Members

Abstract types (e.g., traits) can have **abstract type members**.

It means concrete implementations define their **actual types**.

```scala
trait Box:
  type T
  val elem: T
```

```scala
// Actual type of the abstract type member `T` is `Int`
case class IntBox(elem: Int) extends Box:
  type T = Int
val intBox: IntBox = IntBox(42)   // 42
val intBoxElem: Int = intBox.elem

// Actual type of the abstract type member `T` is `Boolean`
case class BoolBox(elem: Boolean) extends Box:
  type T = Boolean
val boolBox: BoolBox = BoolBox(true)
val boolBoxElem: Boolean = boolBox.elem  // true
```

# Abstract Type Members – Type Bounds

We can also use **type bounds** for abstract type members.

```
trait SeqBox extends Box:
  type Data
  type T <: Seq[Data]
  def length: Int = elem.length
```

```
case class IntListBox(elem: List[Int]) extends SeqBox:
  type Data = Int
  type T = List[Int]
val intListBox: IntListBox = IntListBox(List(1, 3, 2, 6))
val intListBoxElem: List[Int] = intListBox.elem     // List(1, 3, 2, 6)
val intListBoxLen: Int = intListBox.length          // 4
```

```
case class StrVecBox(elem: Vector[String]) extends SeqBox:
  type Data = String
  type T = Vector[String]
val strVecBox: StrVecBox = StrVecBox(Vector("a", "b"))
val strVecBoxElem: Vector[String] = strVecBox.elem  // Vector("a", "b")
val strVecBoxLen: Int = strVecBox.length            // 2
```

# Abstract Type Members – Type Parameters

It is also possible to turn abstract type members into type parameters.

```scala
trait Box[+T]:
  val elem: T

trait SeqBox[Data, +T <: Seq[Data]] extends Box[T]:
  def length: Int = elem.length
```

```scala
case class IntListBox(elem: List[Int])
  extends SeqBox[Int, List[Int]]
val intListBox: IntListBox = IntListBox(List(1, 3, 2, 6))
val intListBoxElem: List[Int] = intListBox.elem     // List(1, 3, 2, 6)
val intListBoxLen: Int = intListBox.length          // 4
```

```scala
case class StrVecBox(elem: Vector[String])
  extends SeqBox[String, Vector[String]]
val strVecBox: StrVecBox = StrVecBox(Vector("a", "b"))
val strVecBoxElem: Vector[String] = strVecBox.elem  // Vector("a", "b")
val strVecBoxLen: Int = strVecBox.length            // 2
```

# Contents

# Inner Classes

In Scala, a class has other classes as members called **inner classes**.

```
class Graph:
  case class Node(id: Int)
  private var nodes: Set[Node] = Set()
  private var edges: Set[(Node, Node)] = Set()
  def allNodes: Set[Node] = nodes
  def allEdges: Set[(Node, Node)] = edges
  def newNode: Node =
    val node = Node(nodes.map(_.id).maxOption.getOrElse(0) + 1)
    nodes += node
    node
  def drawEdge(from: Node, to: Node): Unit = edges += (from, to)
```

```
val graph: Graph = Graph()
val node1 = graph.newNode      // Node(1)
val node2 = graph.newNode      // Node(2)
val node3 = graph.newNode      // Node(3)
graph.drawEdge(node1, node2)
graph.drawEdge(node1, node3)   // graph = 2 <- 1 -> 3
```

# Inner Classes – Path Dependent Types

The inner class `Node` is a **path-dependent type** of the outer class `Graph`.

```
val graphA: Graph = Graph()
val nodeA1: graphA.Node = graphA.newNode    // graphA.Node(1)
val nodeA2: graphA.Node = graphA.newNode    // graphA.Node(2)

// Possible: `nodeA1` and `nodeA2` are nodes of `graphA`
graphA.drawEdge(nodeA1, nodeA2)
```

```
val graphB: Graph = Graph()
val nodeB1: graphB.Node = graphB.newNode    // graphB.Node(1)
val nodeB2: graphB.Node = graphB.newNode    // graphB.Node(2)

// Possible: `nodeB1` and `nodeB2` are nodes of `graphB`
graphB.drawEdge(nodeB1, nodeB2)
```

It means nodes of a graph are **incompatible** with nodes of another graph.

```
// Type Mismatch Error: `graphA.Node` expected but `graphB.Node` found
graphA.drawEdge(nodeA1, nodeB2)
```

# Inner Classes – Path Dependent Types

We can represent types for **inner classes** without depending on the outer class using # symbol.

```
val graphA: Graph = Graph()
val node1: Graph#Node = graphA.newNode    // graphA.Node(1)

val graphB: Graph = Graph()
val node2: Graph#Node = graphB.newNode    // graphB.Node(1)
```

We can also define **path-dependent types** for **abstract member types**.

```
trait Box:
  type T
  val elem: T

case class IntBox(elem: Int) extends Box { type T = Int }
val intBox: Box = IntBox(42)
val intElem: intBox.T = intBox.elem   // 42
```

# Inner Classes – Path Dependent Types

Dependent method/function types are function types whose return type depends on its parameter values using **path-dependent types**.

```
trait Box:
  type T
  val elem: T

def getElem(box: Box): box.T = box.elem
```

The `getElem` function has a **dependent method type** whose return type `box.T` depends on the parameter value `box` of type `Box`.

```
case class IntBox(elem: Int) extends Box { type T = Int }
val intBox: Box = IntBox(42)
val intElem: intBox.T = getElem(intBox)   // 42
```

# Summary

1. Generic Classes
2. Generic Methods/Functions
3. Type Bounds
4. Variances
5. Abstract Type Members
6. Inner Classes

# Next Lecture

- Advanced Types

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