# Lecture 13 – Course Review

## SWS121: Secure Programming

Jihyeok Park

PLRG

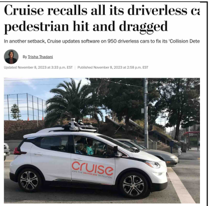
2024 Spring

# Errors in Safety-Critical Software

PLRG

Unexpected faults in **safety-critical software** cause serious problems:




| **Rocket** | **Financial** | **Airport** | **Auto. Vehicle** |
|---|---|---|---|
| (1996) | (2012) | (2020) | (2023) |

Then, how can we **prevent** such software faults?

Let's learn **secure programming** to write **safe** and **reliable** software with **Scala**.

# Goal of This Course

**Secure Programming** is a coding practice that ensures the software is designed to be secure and free from vulnerabilities.

- **Static type checking**
  - Using the type system to catch bugs
- **Test-driven development (TDD)**
  - Writing tests before writing the code
- **Documentation**
  - Writing clear and concise comments
- **Encapsulation**
  - Hiding the implementation details
- **Defensive programming**
  - Writing code to handle unexpected inputs

# What is Scala?

Scala stands for **Sca**lable **La**nguage.

- A **more concise** version of Java with **advanced features**
- A general-purpose programming language
- **Java Virtual Machine (JVM)**-based language
- A **statically typed** language
- An **object-oriented programming (OOP)** language
- A **functional programming (FP)** language

# Schedule

| Weak | Date | Contents |
|---|---|---|
| 1 | 03/04 | Introduction |
| 2 | 03/11 | Basics |
| 3 | 03/18 | Testing and Documentation |
| 4 | 03/25 | Classes, Traits, and Objects |
| 5 | 04/01 | Functional Programming |
| 6 | 04/08 | Immutable Collections |
| 7 | 04/15 | For Comprehensions |
| 8 | 04/22 | Midterm Exam Week (No Class) |
| 9 | 04/29 | Lazy Evaluation |
| 10 | 05/06 | Generics |
| 11 | 05/13 | Advanced Types |
| 12 | 05/20 | Contextual Abstraction |
| 13 | 05/27 | Metaprogramming |
| 14 | 06/03 | Concurrent Programming |
| 15 | 06/10 | Course Review |
| 16 | 06/17 | Final Exam Week (No Class) |

# 1. Basics

- **Basic Features**

```
def sum(n: Int): Int = if (n < 1) 0 else sum(n - 1) + n
sum(100)                       // 5050      : Int
```

- **Algebraic Data Types (ADTs)**

```
enum Nat:
  case Zero
  case Succ(n: Nat)
```

- **First-Class Functions**

```
def twice(f: Int => Int, x: Int): Int = f(f(x))
val addN = (n: Int) => (x: Int) => x + n
twice(addN(7), 5)             // 5 + 7 + 7 = 19: Int
```

- **Immutable Collections** (e.g., `List`, `Set`, `Map`)

# 2. Testing and Documentation

- **Simple Build Tool (`sbt`) for Scala**
  - `sbt` is a **simple build tool** for Scala and Java projects. It is similar to Maven or Ant, but it is designed for **Scala**.
- **Scala Documentation**
  - `scaladoc` **automatically generates documentation** from **comments** in Scala source code.
- **Scala Test Framework**
  - `ScalaTest` is a **test framework** for Scala and Java Virtual Machine (JVM) that is designed to be **scalable** and **flexible**.
  - We can **measure the code coverage** of the project using `scoverage`, the **code coverage tool** for Scala.

# 3. Classes, Traits, and Objects (1)

- **Constructors**

```
case class Person(name: String, age: Int)
val p1 = Person("Jihyeok Park", 32)
val p2 = p1.copy(age = 50)  // Person("Jihyeok Park", 50)
```

- **Traits**

```
trait HasName { val name: String; def hi = s"Hi, $name!" }
trait HasLegs { def legs: Int; def walk = s"Have $legs legs" }
trait NamedTwoLegs extends HasName, HasLegs { def legs = 2 }
case class Person(name: String, age: Int) extends NamedTwoLegs
val p = Person("Jihyeok", 32)
p.hello     // "Hi, Jihyeok!"
p.walk      // "Have 2 legs"
```

- **Overloading** and **Overriding** (e.g., `super`, linearization)
- **Access Modifiers** (e.g., `private`, `protected`)

# 3. Classes, Traits, and Objects (2)

- **Objects**, **Companion Objects**, and **Operators**

```
case class Point(x: Int, y: Int):
  def +(that: Point): Point = Point(this.x + that.x, this.y + that.y)
  def *(k: Int): Point = Point(this.x * k, this.y * k)

object Point:
  def apply(k: Int): Point = Point(k, k)

val p1 = Point(2)
val p2 = Point(3, 4)
p1 + p2 * 2    // Point(2, 2) + Point(3, 4) * 2 = Point(7, 10)
```

# 4. Functional Programming (1)

- **Functions**
  - **Methods vs Functions**

```scala
def isEvenMethod(n: Int) = n % 2 == 0             // a method
val isEvenFunction = (n: Int) => n % 2 == 0       // a function
```

  - **Tail-Call Optimization** and **Nested Methods**

```scala
def sum(n: Int): Int = {
  @tailrec
  def aux(n: Int, acc: Int): Int =
    if (n < 1) acc
    else aux(n - 1, n + acc) // tail-call
  aux(n, 0)
```

  - **Multiple Parameter Lists**

```scala
def addN(x: Int)(y: Int): Int = x + y
val add3 = addN(3)
add3(5)        // 3 + 5 = 8
```

# 4. Functional Programming (2)

- **Pattern Matching**
  - **Exhaustive Matching** and **Reachability**

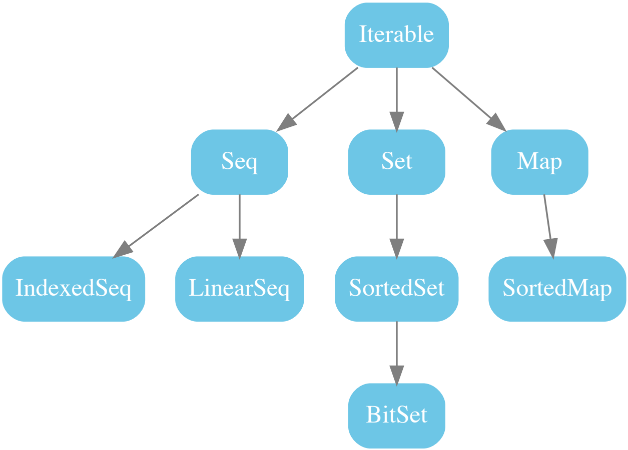
```
val minMerge: (List[Int], List[Int]) => List[Int] = {
  case (x :: l1, y :: l2) if x < y => x :: merge(l1, l2)
  case (Nil, _) | (_, Nil) => Nil
  case (Nil, Nil) => Nil // unreachable
} // missing for (x :: l1, y :: l2) if x >= y
```

  - **Regular Expression Patterns** and **Extractor Objects**

```
object Pair:
  val pairPattern = "\\((\\d+), *(\\d+)\\)".r
  def unapply(s: String): Option[(Int, Int)] = s match
    case pairPattern(x, y) => Some(x.toInt, y.toInt)
    case _ => None
"(1, 2)" match
  case Pair(x, y) => x + y // 1 + 2 = 3
```

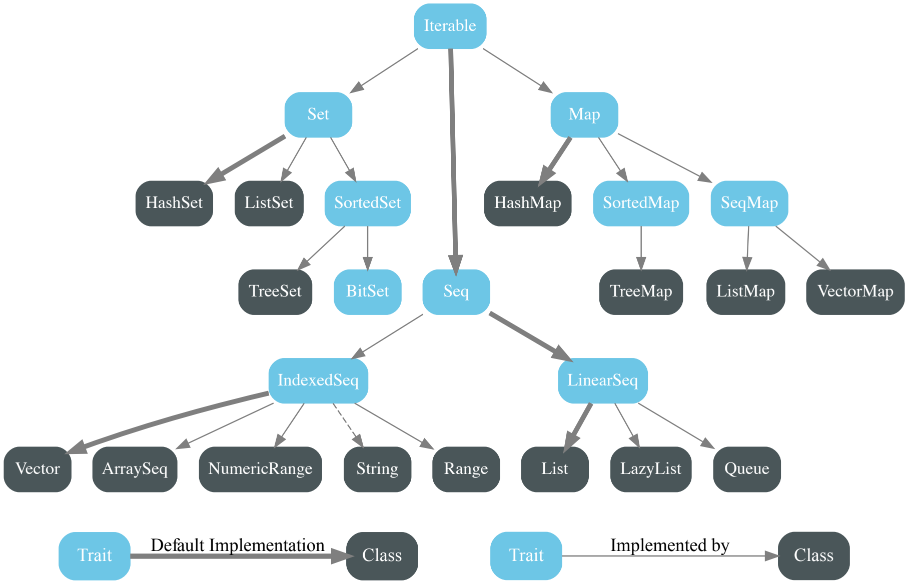
- **Functional Error Handling** (e.g., `Option`, `Try`, `Either`)

# 5. Immutable Collections




- **Immutable Collections** for **Thread Safety**, **Security**, **Easier Debugging**, **Memory Efficiency**

# 5. Immutable Collections

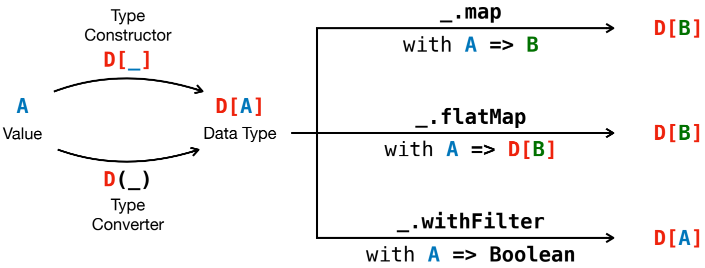


# 6. For Comprehensions (1)

- **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
```



# 6. For Comprehensions (2)

- 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

# 7. Lazy Evaluation

- **Lazy Evaluation** for **Performance** and **Readability**
- **Call-By-Need – Caching** (e.g., `lazy val`)

```
lazy val x: Int = { Thread.sleep(5000); 42 }
if (...)
  x   // 42 (after 5 seconds)
  x   // 42 (immediately)
else ... // no evaluation of `x`
```

- **Call-By-Name – No Caching** (e.g., =>)

```
def f(y: => Int): Int = y + y   // 84 (after 5+5 seconds)
f({ Thread.sleep(5000); 42 })
```

- **Lazy Lists**

```
lazy val fib: LazyList[Int] =       // Infinite Fibonacci sequence
  1 #:: 1 #:: (fib zip fib.tail).map(_ + _)
```

# 8. Generics

- **Generic Classes/Methods/Functions**

```scala
case class Box[A](value: A):
  def map[B](f: A => B): Box[B] = Box(f(value))
def get[A](box: Box[A]): A = box.value
get(Box("abc").map(_.length))  // 3: Int
```

- **Type Bounds** (e.g., <:) and **Variance** (e.g., +, −)

```scala
case class Box[+A <: Seq[Int]](value: A)
val box1: Box[Seq[Int]] = Box[List[Int]](List(1, 2, 3))
val box2: Box[Boolean] = ??? // Upper bound violation <: Seq[Int]
```

- **Abstract Type Members** and **Inner Classes**

```scala
trait HasData { type Data }
class Graph { case class Node(id: Int); type Data = Node }
val graph1: Graph = Graph()
val node1: graph1.Node = graph1.Node(1)
val data1: graph1.Data = node1
```

# 9. Advanced Types (1)

- **Intersection and Union Types**

```scala
trait A { def foo(x: Int): Int }
trait B { def bar(x: Int): Int }
def f(x: A & B): Int = x.foo(10) + x.bar(20)
val x: Int | String = "abc"
```

- **Self Types**

```scala
trait User { def username: String }
trait Tweeter { this: User => }
case class VerifTweeter(username: String) extends User with Tweeter
```

- **Opaque Types**

```scala
object Logarithms:
  opaque type Logarithm = Double
val log: Logarithms.Logarithm = 10.0 // Error: type mismatch
```

# 9. Advanced Types (2)

- **Structural Types**

```scala
class Duck { def fly = println("Duck flies") }
val flyable: { def fly: Unit } = Duck()
```

- **Type Lambdas**

```scala
type MapInt = [X] =>> Map[Int, X]
val m1: MapInt[String] = Map(1 -> "one", 2 -> "two")
```

- **Polymorphic Function Types**

```scala
val id: [A] => A => A = [A] => (x: A) => x
val idInt: Int => Int = id[Int]
```

- **Match Types**

```scala
type Elem[X] = X match { case String => Char; case List[t] => t }
def firstElem[X](xs: X): Elem[X] = xs match
  case x: String    => x.charAt(0)
  case x: List[t]   => x.head
```

# 10. Contextual Abstraction

- **Context Parameters**

```scala
def show(msg: String)(using pre: String): String = s"[$pre] $msg"
given String = "info";    show("Hello") // "[info] Hello"
```

- **Implicit Conversions**

```scala
given Conversion[String, Int] = _.length
val length: Int = "abc" // 3
```

- **Extension Methods**

```scala
extension (s: Int) def double: Int = s * 2;     42.double // 84
```

- **Type Classes**

```scala
trait Show[A] { extension (a: A) def show: String }
given Show[Int] with
  extension (n: Int) { def show: String = "a" * n }
def showAll[A: Show](xs: List[A]): List[String] = xs.map(_.show)
showAll(List(1, 2, 3)) // List("a", "aa", "aaa")
```

# 11. Metaprogramming

- **Inline Constants**

```
inline val Pi = 3.14
2 * Pi * radius // compiled to `6.28 * radius`
```

- **Inline Methods and Parameters**

```
inline def add(inline x: Int, inline y: Int): Int = x + y
add(a * 2, b * 3) // compiled to `(a * 2) + (b * 3)`
```

- **Inline Matches** and **Transparent Inline Methods**

```
transparent inline def evenOrOdd(inline n: Int): String =
  inline (n % 2) match
    case 0 => "even"
    case 1 => "odd"
val x: "even" = evenOrOdd(42)
```

- **Macros** (e.g., `Expr[T]`, `Quotes`)

- **Futures**
  - **Callbacks** – `onComplete`, `onSuccess`, `onFailure`
  - **Combinators** – `map`, `flatMap`, `filter`, `foreach`
  - **Multiple Futures**

```
val f = Future { Thread.sleep(1_000); 5 }
val g = Future { Thread.sleep(2_000); 6 }
val h = Future { Thread.sleep(3_000); 7 }
val result = for { x <- f; y <- g; z <- h } yield x + y + z
result.foreach { r => println(r) } // 18 after 3 seconds
```

- **Promise** – a writable, single-assignment container, which completes a future with a value using `success` or `failure` methods.
- **Parallel Collection**

```
def slowInc(x: Int): Int = { Thread.sleep(1_000); x + 1 }
val list = List(1, 2, 3, 4, 5)
list.par.map(slowInc).toList // List(2, 3, 4, 5, 6) after 1 second
```

- I hope you enjoyed the class!

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