Lecture 3 – Classes, Traits, and Objects

SWS121: Secure Programming

Jihyeok Park

PLRG

2024 Spring

Recall

- Simple Build Tool (`sbt`) for Scala
  - Example Project
  - Project Structure
  - Building a Project
  - Running a Project
- Scala Documentation
  - `scaladoc` – Scala Documentation Tool
  - Generating Documentation
  - Writing Documentation
- Scala Test Framework
  - Why Software Testing?
  - `ScalaTest` – Test Framework for Scala
  - Running Tests
  - Writing Tests
  - Measuring Code Coverage

Contents

Contents

Recall: Product Types – Case Classes

PLRG

A `case class` defines a **product type** with named fields.

type name **field type**

```
case class Point(x: Int, y: Int, color: String)
```

field name

```
// A case class `Point` having `x`, `y`, and `color` fields
// whose types are `Int`, `Int`, and `String`, respectively
case class Point(x: Int, y: Int, color: String)

// A `Point` instance whose fields: x = 3, y = 4, and color = "RED"
val point: Point = Point(3, 4, "RED")

// You can access fields using the dot operator
point.x            // 3       : Int
point.color        // "RED"   : String

// Fields are immutable by default
point.x = 5        // Type Error: Reassignment to val `x`
```

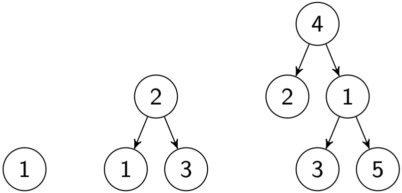
Recall: Algebraic Data Types (ADTs) – Enumerations

An **algebraic data type (ADT)** is a sum of product types, and you can define it using **enumerations** (`enum`) in Scala.

type name (`Tree`) — **variants** (the `case` lines)

```
enum Tree:
  case Leaf(value: Int)
  case Branch(left: Tree, value: Int, right: Tree)
```

```
import Tree.* // Import all constructors for variants of `Tree`
val tree1: Tree = Leaf(1)
val tree2: Tree = Branch(Leaf(1), 2, Leaf(3))
val tree3: Tree = Branch(Leaf(2), 4, Branch(Leaf(3), 1, Leaf(5)))
```



Contents

Constructors – Auxiliary Constructors

A `case class` has a default constructor.

We can define **auxiliary constructors** using the `this` keyword.

Constructors – Auxiliary Constructors

A `case class` has a default constructor.

We can define **auxiliary constructors** using the `this` keyword.

```
case class Person(name: String, age: Int):
  def this(firstName: String, lastName: String, age: Int) =
    this(s"$firstName $lastName", age)

val p1 = Person("Jihyeok Park", 32)
val p2 = new Person("Jihyeok", "Park", 32)
p1 == p2    // true
```

Constructors – Auxiliary Constructors

A `case class` has a default constructor.

We can define **auxiliary constructors** using the `this` keyword.

```
case class Person(name: String, age: Int):
  def this(firstName: String, lastName: String, age: Int) =
    this(s"$firstName $lastName", age)

val p1 = Person("Jihyeok Park", 32)
val p2 = new Person("Jihyeok", "Park", 32)
p1 == p2    // true
```

```
case class Person(name: String, age: Int):
  ...
  def this() = this("Unknown", 0)

val p3 = Person("Unknown", 0)
val p4 = new Person()
p3 == p4    // true
```

Constructors – The `copy` Method

Instead of constructors, we can use the `copy` method for a `case class` instance to create a new instance with some fields modified.

Constructors – The copy Method

Instead of constructors, we can use the copy method for a `case class` instance to create a new instance with some fields modified.

```
val p1 = Person("Jihyeok Park", 32)
val p2 = p1.copy(age = 50)
p2 == Person("Jihyeok Park", 50)    // true

val p3 = p1.copy(name = "Unknown")
p3 == Person("Unknown", 32)         // true
```

Constructors – The copy Method

Instead of constructors, we can use the copy method for a `case class` instance to create a new instance with some fields modified.

```
val p1 = Person("Jihyeok Park", 32)
val p2 = p1.copy(age = 50)
p2 == Person("Jihyeok Park", 50)    // true

val p3 = p1.copy(name = "Unknown")
p3 == Person("Unknown", 32)         // true
```

Note that the copy method does **not modify** the original instance.

It creates a **new instance** with the specified fields modified.

Constructors – The copy Method

Instead of constructors, we can use the copy method for a `case class` instance to create a new instance with some fields modified.

```
val p1 = Person("Jihyeok Park", 32)
val p2 = p1.copy(age = 50)
p2 == Person("Jihyeok Park", 50)    // true

val p3 = p1.copy(name = "Unknown")
p3 == Person("Unknown", 32)         // true
```

Note that the copy method does **not modify** the original instance.

It creates a **new instance** with the specified fields modified.

And, it utilizes the **named arguments** feature in Scala.

```
def f(x: Int, y: Int): Int = x + y
f(1, 2)             // 3
f(x = 1, y = 2)     // 3
f(y = 2, x = 1)     // 3
```

Traits

A `trait` is similar to an interface in Java.

It defines a **type** with specific **abstract** or **concrete** methods and fields.

```
trait HasName:
  // Abstract field
  val name: String
  // Concrete method
  def hello: String = s"Hello, $name!"

trait HasLegs:
  // Abstract method
  def numLegs: Int
  // Concrete method
  def walk: String = s"Walking on $numLegs legs"
```

Traits – Extending and Implementing

We can define a class that **extends** one or more traits.

We need to implement all **abstract** methods and fields.

Traits – Extending and Implementing

We can define a class that **extends** one or more traits.

We need to implement all **abstract** methods and fields.

```
// A class `Person` extending `HasName`
case class Person(name: String, age: Int) extends HasName, HasLegs:
  def numLegs: Int = 2

val p = Person("Jihyeok", 32)
p.name      // "Jihyeok"              -- abstract field in `HasName`
p.hello     // "Hello, Jihyeok!"      -- concrete method in `HasName`
p.numLegs   // 2                      -- abstract method in `HasLegs`
p.walk      // "Walking on 2 legs"    -- concrete method in `HasLegs`
```

Traits – Extending and Implementing

PLRG

We can define a class that **extends** one or more traits.

We need to implement all **abstract** methods and fields.

```
// A class `Person` extending `HasName`
case class Person(name: String, age: Int) extends HasName, HasLegs:
  def numLegs: Int = 2

val p = Person("Jihyeok", 32)
p.name      // "Jihyeok"             -- abstract field in `HasName`
p.hello     // "Hello, Jihyeok!"     -- concrete method in `HasName`
p.numLegs   // 2                     -- abstract method in `HasLegs`
p.walk      // "Walking on 2 legs"   -- concrete method in `HasLegs`
```

The type `Person` is a **subtype** of `HasName` and `HasLegs`.

Therefore, the variable `p` can be a `HasName` or `HasLegs`.

```
val hasName: HasName = p
val hasLegs: HasLegs = p
```

Traits – Mixin Composition

We can define a new trait by **mixing** multiple traits.

Traits – Mixin Composition

We can define a new trait by **mixing** multiple traits.

```
trait HasName:
  val name: String
  def hello: String = s"Hello, $name!"

trait HasLegs:
  def numLegs: Int
  def walk: String = s"Walking on $numLegs legs"
```

For example, NamedTwoLegged mixes HasName and HasLegs traits.

```
trait NamedTwoLegged extends HasName, HasLegs:
  def numLegs: Int = 2      // implement `numLegs` in `HasLegs`

case class Person(name: String, age: Int) extends NamedTwoLegged

val p = Person("Jihyeok", 32)
p.hello     // "Hello, Jihyeok!"
p.walk      // "Walking on 2 legs"
```

Overloading

We can define multiple methods with the same name but different numbers or types of parameters.

It is called **method overloading**.

Overloading

We can define multiple methods with the same name but different numbers or types of parameters.

It is called **method overloading**.

```scala
case class A():

  def f(x: Int): Int = x

  // Overloaded method with different number of parameters
  def f(x: Int, y: Int): Int = x + y

  // Overloaded method with different types of parameters
  def f(x: String): String = x + "!"

val a = A()
a.f(1)          // 1
a.f(1, 2)       // 1 + 2 = 3
a.f("Hello")    // "Hello" + "!" = "Hello!"
```

Overriding

We can define a method in a subclass that has the same signature as a method in its superclass using the `override` keyword.

It is called **method overriding**.

Overriding

We can define a method in a subclass that has the same signature as a method in its superclass using the `override` keyword.

It is called **method overriding**.

```
trait Animal:
  def speak: String = "Animal speaks"

class Dog extends Animal:
  override def speak: String = "Dog barks"

Dog().speak   // "Dog barks"
```

Overriding

We can define a method in a subclass that has the same signature as a method in its superclass using the `override` keyword.

It is called **method overriding**.

```
trait Animal:
  def speak: String = "Animal speaks"

class Dog extends Animal:
  override def speak: String = "Dog barks"

Dog().speak   // "Dog barks"
```

We can prevent a method from being overridden by using `final` modifier.

```
trait Animal:
  final def speak: String = "Animal speaks"

class Dog extends Animal:
  override def speak: String = "Dog barks"    // Compile Error
```

Overriding – The `super` Keyword

We can call the overridden method in the superclass using the `super` keyword.

Overriding – The super Keyword

We can call the overridden method in the superclass using the `super` keyword.

```
trait Animal:
  def speak: String = "Animal speaks"

trait Dog extends Animal:
  override def speak: String =
    super.speak + " and Dog barks"

class Puppy extends Dog:
  override def speak: String =
    super.speak + " and Puppy whines"

Puppy().speak   // "Animal speaks and Dog barks and Puppy whines"
```

Overriding – Diamond Problem

In Java, a class can extend **only one class**, but it can implement **multiple interfaces** (no real implementation – all abstract methods).

It is due to the **diamond problem** in multiple inheritance.

If Java allows multiple inheritance for classes, it may cause ambiguity:

Overriding – Diamond Problem

In Java, a class can extend **only one class**, but it can implement **multiple interfaces** (no real implementation – all abstract methods).

It is due to the **diamond problem** in multiple inheritance.

If Java allows multiple inheritance for classes, it may cause ambiguity:

```
class Parent1 {
  void fun() { System.out.println("Parent1"); }
}
class Parent2 {
  void fun() { System.out.println("Parent2"); }
}
class test extends Parent1, Parent2 { }

test t = new test();
t.fun();       // `fun` method is ambiguous (Parent1 or Parent2)
```

Overriding – Linearization

However, Scala solves this problem using **linearization**.

When a class extends multiple traits having the same concrete method, Scala uses the **rightmost** trait's method.

Overriding – Linearization

However, Scala solves this problem using **linearization**.

When a class extends multiple traits having the same concrete method, Scala uses the **rightmost** trait's method.

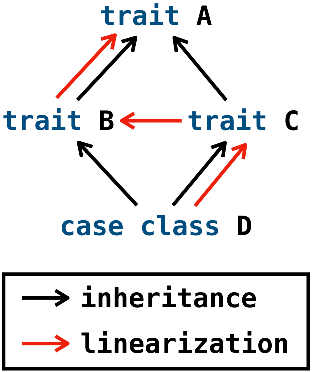
```scala
trait A:
  def f: Int = 0

trait B extends A:
  override def f: Int = 1

trait C extends A:
  override def f: Int = 2

case class D() extends B, C

D().f          // 2
```



Overriding – Linearization

However, Scala solves this problem using **linearization**.

When a class extends multiple traits having the same concrete method, Scala uses the **rightmost** trait's method.

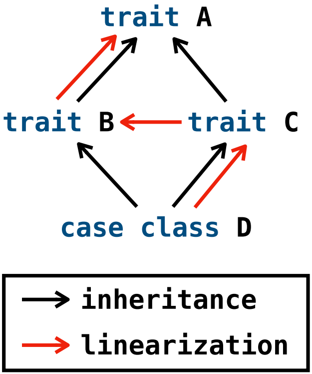
```
trait A:
  def f: Unit = println("A")

trait B extends A:
  override def f: Unit =
    super.f; println("-> B")

trait C extends A:
  override def f: Unit =
    super.f; println("-> C")

case class D() extends B, C:
  override def f: Unit =
    super.f; println("-> D")

D().f   // A -> C -> B -> D
```



Access Modifiers

PLRG

Similar to Java, Scala provides **access modifiers**: `private` and `protected` to restrict access to fields and methods.

Access Modifiers

Similar to Java, Scala provides **access modifiers**: `private` and `protected` to restrict access to fields and methods.

```
trait A:
  val x: Int = 0              // public by default
  protected val y: Int = 1   // protected
  private val z: Int = 0     // private

case class B() extends A:
  def getX: Int = x          // Can access `x` in `A`
  def getY: Int = y          // Can access `y` in `A`
  def getZ: Int = z          // Compile Error: `z` is private in `A`

val b = B()
b.x                          // 0
b.y                          // Compile Error: `y` is protected in `A`
b.z                          // Compile Error: `z` is private in `A`
```

Access Modifiers – Setter Syntax

Scala supports special postfix syntax _= with a field/method name for defining **setters**.

Access Modifiers – Setter Syntax

PLRG

Scala supports special postfix syntax _= with a field/method name for defining **setters**.

```scala
case class A():
  private var _x: Int = 0
  private val BOUND = 100
  // Getter for `_x`
  def x: Int = _x
  // Setter for `_x`
  def x_=(newX: Int): Unit =
    if (newX > BOUND) _x = BOUND
    else              _x = newX

val a = A()
a.x         // 0
a.x = 10    // set `x` to 10
a.x         // 10
a.x = 200   // set `x` to 100 because 200 > 100
a.x         // 100
```

Contents

Objects

In Scala, we can define a singleton object using the `object` keyword without the definition of a class.

Objects

In Scala, we can define a singleton object using the `object` keyword without the definition of a class.

```
object StringUtils:
  def truncate(s: String, length: Int): String = s.take(length)
  def repeat(s: String, n: Int): String = s * n
  def toUpperCase(s: String): String = s.toUpperCase

StringUtils.truncate("Hello, World!", 5)    // "Hello"
StringUtils.repeat("Scala", 3)              // "ScalaScalaScala"
StringUtils.toUpperCase("scala")            // "SCALA"
```

Objects

In Scala, we can define a singleton object using the `object` keyword without the definition of a class.

```
object StringUtils:
  def truncate(s: String, length: Int): String = s.take(length)
  def repeat(s: String, n: Int): String = s * n
  def toUpperCase(s: String): String = s.toUpperCase

StringUtils.truncate("Hello, World!", 5)     // "Hello"
StringUtils.repeat("Scala", 3)               // "ScalaScalaScala"
StringUtils.toUpperCase("scala")             // "SCALA"
```

Or, we can import the methods from the object and use them directly.

```
import StringUtils.*

truncate("Hello, World!", 5)    // "Hello"
repeat("Scala", 3)              // "ScalaScalaScala"
toUpperCase("scala")            // "SCALA"
```

Objects – The `apply` Method

An object can have an `apply` method can be invoked without the method name.

It looks like calling an object as a function.

Objects – The apply Method

An object can have an apply method can be invoked without the method name.

It looks like calling an object as a function.

```
object Square:
  def apply(x: Int): Int = x * x

val x: Int = Square(5)                          // 25

object Concat:
  def apply(s1: String, s2: String): String = s1 + s2

val s: String = Concat("Hello, ", "World!")    // "Hello, World!"
```

Objects – The `apply` Method

You can define the `apply` method in a class as well.

Objects – The apply Method

You can define the `apply` method in a class as well.

```
case class Adder(n: Int):
  def apply(x: Int): Adder = Adder(n + x)
  def calculate(x: Int): Int = x + n

val add5 = Adder(5)                 // Adder(5)
add5.calculate(7)                   // 7 + 5 = 12

val add12 = add5(7)                 // Adder(12)
add12.calculate(3)                  // 3 + 12 = 15

Adder(1)(2)(3)(4).calculate(5)  // Adder(10).calculate(5) = 5 + 10 = 15
```

Companion Objects

Especially, a singleton object with the same name as a class is called a **companion object**.

Similarly, the corresponding class is called a **companion class**.

Companion Objects

Especially, a singleton object with the same name as a class is called a **companion object**.

Similarly, the corresponding class is called a **companion class**.

The companion object **can access the private fields and methods** of the companion class, and vice versa.

```
case class Square(side: Int):
  private def area: Int = side * side
  // Companion class can access private fields in companion object
  def getName: String = Square.name

object Square:
  private val name: String = "Square"
  // Companion object can access private fields in companion class
  def calculateArea(square: Square): Int = square.area

Square(5).getName                // "Square"
Square.calculateArea(Square(5))  // 25
```

Companion Objects

Scala supports such companion objects to define **static** fields and methods shared by all instances of the class like in Java.

Companion Objects

Scala supports such companion objects to define **static** fields and methods shared by all instances of the class like in Java.

For example, we can implement the left Java implementation in Scala using companion objects.

```java
class Counter {
  static int count = 0;
  void increment() {
    count++;
  }
}

Counter c1 = new Counter();
c1.increment();
Counter c2 = new Counter();
c2.increment();
c1.count;            // 2
```

```scala
case class Counter():
  def increment: Unit =
    Counter.count += 1

object Counter:
  var count: Int = 0

Counter().increment
Counter().increment
Counter.count          // 2
```

Companion Objects – The apply Method

PLRG

Using the apply method in the companion object, we can create an instance of the class without the new keyword.

```
case class Person(name: String, age: Int)

object Person:
  def apply(firstName: String, lastName: String, age: Int): Person =
    Person(s"$firstName $lastName", age)
  def apply(): Person = Person("Unknown", 0)

val p1 = Person("Jihyeok Park", 32)
val p2 = Person("Jihyeok", "Park", 32)
p1 == p2    // true

val p3 = Person("Unknown", 0)
val p4 = Person()
p3 == p4    // true
```

Operators

We can define **custom operators** in Scala using the `def` keyword exactly same as a method.

Operators

We can define **custom operators** in Scala using the `def` keyword exactly same as a method.

For example, we can define a + operator for a `Point` class.

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

val p1 = Point(1, 2)
val p2 = Point(3, 4)
p1 + p2                    // Point(4, 6)
```

Operators

We can define **custom operators** in Scala using the `def` keyword exactly same as a method.

For example, we can define a + operator for a `Point` class.

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

val p1 = Point(1, 2)
val p2 = Point(3, 4)
p1 + p2                    // Point(4, 6)
```

For unary operators, we need to define a method with a prefix `unary_`:

```
case class Point(x: Int, y: Int):
  ...
  def unary_- : Point = Point(-x, -y)

val p = Point(1, 2)
-p                          // Point(-1, -2)
```

Operators – Infix Notation

We can use the **infix notation** also for methods taking one argument.

```
case class Point(x: Int, y: Int):

  // Additive operator
  def +(that: Point): Point = Point(this.x + that.x, this.y + that.y)

  // A method to find the minimum point
  def min(that: Point): Point =
    if (this.x <= that.x && this.y <= that.y) this
    else                                      that

val p1 = Point(1, 2)
val p2 = Point(3, 4)

// Infix notation for operators
p1 + p2             // Point(4, 6)

// Infix notation for normal methods
p1 min p2           // Point(1, 2)
```

Operators – Precedence

Scala has a set of **operator precedence** rules, and it is also applied to custom operators.

```scala
case class Point(x: Int, y: Int):

  // Additive operator
  def +(that: Point): Point = Point(this.x + that.x, this.y + that.y)

  // Multiplicative operator
  def *(k: Int): Point = Point(this.x * k, this.y * k)

  // Comparison operator
  def <=(that: Point): Boolean = this.x <= that.x && this.y <= that.y

val p1 = Point(1, 2)
val p2 = Point(3, 4)
val p3 = Point(7, 10)

p1 + p2 * 2 <= p3       // (p1 + (p2 * 2)) <= p3 -- true
```

Summary

1. Recall: Product Types and Algebraic Data Types

2. Basic Object-Oriented Programming
    - Constructors
    - Traits
    - Overloading and Overriding
    - Access Modifiers

3. Advanced Object-Oriented Programming
    - Objects
    - Companion Objects
    - Operators

Next Lecture

- Functional Programming

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