Functional Programming in Scala

Classes, Objects, and Traits

Said BOUDJELDA

Eferi

May 4, 2025

Scala 3 OOP Overview

- **Classes**: Templates for object creation
- **Objects**: Singleton instances
- **Traits**: Reusable behavior components

```scala
// Class example
class Person(val name: String)

// Object example
object Logger:
  def log(msg: String) = println(msg)

// Trait example
trait Speaker:
  def speak(): String
```

Scala 3 vs Scala 2 Improvements

- **Simpler syntax**:

```
// Old
class Foo { ... }
// New
class Foo: ...
```

- **New modifiers**:

```
open class Parent  // Explicitly extensible
transparent trait Logger  // More precise type
    checking
```

- **Trait parameters**:

```
trait Config(env: String)  // New in Scala 3
```

Class Constructors

Primary Constructor

```
class Person(
  val name: String,          // Public read-only field
  var age: Int,              // Public mutable field
  private val id: Int        // Private field
):
  def this(name: String) = this(name, 0, 0)  //
      Auxiliary
```

- Parameters in class declaration become constructor parameters
- `val` creates immutable field, `var` mutable
- Auxiliary constructors must call primary constructor

Class Inheritance

```scala
open class Animal(val name: String):
  def makeSound(): String = "Some sound"

class Dog(name: String) extends Animal(name):
  override def makeSound() = "Woof!"
  def fetch() = "Fetching..."
```

Key points:

- `open` required for extension (new in Scala 3)
- `override` modifier required
- Single inheritance only (use traits for multiple)

Case Classes

```scala
case class Person(
  name: String,
  age: Int,
  address: Address = Address.default
)

// Automatically gets:
// 1. equals/hashCode
// 2. toString
// 3. copy method
// 4. companion object with apply
// 5. pattern matching support
```

Usage examples:

```scala
val p1 = Person("Alice", 30)
val p2 = p1.copy(age = 31)  // Non-destructive update
```

Abstract Classes

```scala
abstract class Shape(val color: String):
  def area: Double       // Abstract method
  def perimeter: Double // Abstract method
  def describe = s"$color shape"  // Concrete method

class Circle(color: String, radius: Double)
  extends Shape(color):
    def area = math.Pi * radius * radius
    def perimeter = 2 * math.Pi * radius
```

Characteristics:

- Cannot be instantiated
- Can contain both abstract and concrete members
- Single inheritance still applies

Singleton Objects

```
object MathConstants:
  val PI = 3.1415926535
  val E = 2.7182818284
  def square(x: Double) = x * x

// Usage:
MathConstants.PI
MathConstants.square(5)
```

Key uses:

- Constants and utility methods
- Factory methods
- Entry points for applications

Companion Objects

```scala
class BankAccount private (val balance: Double):
  // Instance members here

object BankAccount:  // Companion
  def apply(initial: Double) =
    new BankAccount(initial)

  def fromString(s: String): Option[BankAccount] =
    s.toDoubleOption.map(new BankAccount(_))
```

Benefits:

- Access to private class members
- Logical grouping of factory methods
- Alternative constructors

Case Class Companions

```scala
case class Email(user: String, domain: String)

// Generated companion includes:
object Email:
  // Factory method
  def apply(user: String, domain: String) =
    new Email(user, domain)

  // Extractor for pattern matching
  def unapply(email: Email): Option[(String, String)]
    =
    Some((email.user, email.domain))
```

Usage:

```scala
Email("user", "example.com")  // No 'new' needed
email match
  case Email(u, d) => println(s"User: $u, Domain: $d")
```

Object Inheritance

```
trait JsonSerializer:
  def toJson: String

object DefaultSerializer extends JsonSerializer:
  def toJson = "{}"

// Objects can extend classes/traits
// But cannot be extended themselves
```

Limitations:

- Objects are final (cannot be extended)
- Can mix in multiple traits
- Useful for implementing type classes

Trait Basics

```scala
trait Logger:
  def log(msg: String): Unit  // Abstract method
  def info(msg: String) = log(s"INFO: $msg")  //
      Concrete
  def warn(msg: String) = log(s"WARN: $msg")  //
      Concrete

class ConsoleLogger extends Logger:
  def log(msg: String) = println(msg)  // Implement
      abstract
```

Characteristics:

- Can contain abstract and concrete members
- Multiple inheritance allowed
- Cannot have constructor parameters (pre-Scala 3)

Trait Parameters (Scala 3)

```scala
trait Greeting(val prefix: String):
  def greet(name: String) = s"$prefix $name"

class FormalGreeter extends Greeting("Dear")
class CasualGreeter extends Greeting("Hey")

// Usage:
FormalGreeter().greet("Alice")  // "Dear Alice"
CasualGreeter().greet("Bob")    // "Hey Bob"
```

Advantages:

- Parameterized behavior without abstract members
- Evaluated exactly once when mixed in
- Alternative to constructor parameters in traits

Stackable Traits Pattern

```
abstract class IntQueue:
  def get(): Int
  def put(x: Int): Unit

trait Doubling extends IntQueue:
  abstract override def put(x: Int) =
    super.put(2 * x)

trait Incrementing extends IntQueue:
  abstract override def put(x: Int) =
    super.put(x + 1)

class BasicQueue extends IntQueue:
  private val buf = collection.mutable.ArrayBuffer.
      empty[Int]
  def get() = buf.remove(0)
  def put(x: Int) = buf += x
```

Stackable Traits Usage

```
val q1 = new BasicQueue with Doubling
q1.put(10)  // Adds 20
q1.get()    // Returns 20

val q2 = new BasicQueue with Incrementing
q2.put(10)  // Adds 11
q2.get()    // Returns 11

val q3 = new BasicQueue with Doubling with
    Incrementing
q3.put(10)  // Adds 22 (10*2 then +1)
q3.get()    // Returns 22

val q4 = new BasicQueue with Incrementing with
    Doubling
q4.put(10)  // Adds 21 (10+1 then *2)
q4.get()    // Returns 21
```

Self Types

```
trait UserRepository:
  def findUser(id: Int): User

trait UserService { self: UserRepository =>  // Self-
    type
  def getUser(id: Int): User =
    findUser(id).orElse(defaultUser)
}

// Must mix in UserRepository
class UserServiceImpl extends UserService with
    UserRepository:
  def findUser(id: Int) = ...
```

Purpose:

- Declare trait dependencies without inheritance
- Enforce required mixins
- Avoid circular dependencies

Type Class Pattern

```scala
// 1. Type class definition
trait JsonWriter[A]:
  def write(value: A): Json

// 2. Type class instances
object JsonWriterInstances:
  given JsonWriter[String] with
    def write(s: String) = JsonString(s)

  given JsonWriter[Int] with
    def write(n: Int) = JsonNumber(n)

// 3. Interface
object Json:
  def toJson[A](value: A)(using writer: JsonWriter[A])
      =
    writer.write(value)
```

Extension Methods

```scala
trait StringExtensions:
  extension (s: String)
    def toTitleCase: String =
      s.split(" ").map(_.capitalize).mkString(" ")

    def encrypt(shift: Int): String =
      s.map(c => (c + shift).toChar)

// Usage:
import StringExtensions.*
"hello world".toTitleCase  // "Hello World"
"abc".encrypt(1)           // "bcd"
```

Benefits:

- Add methods to existing types
- More discoverable than implicit classes
- Group related extensions together

Factory Pattern with Companion

```scala
sealed abstract class DatabaseConfig

object DatabaseConfig:
  // Private implementations
  private case class PostgresConfig(url: String)
    extends DatabaseConfig

  private case class MongoConfig(uri: String)
    extends DatabaseConfig

  // Factory methods
  def postgres(url: String): DatabaseConfig =
    PostgresConfig(url)

  def mongo(uri: String): DatabaseConfig =
    MongoConfig(uri)
```

Value Classes

```scala
class Meter(val value: Double) extends AnyVal:
  def +(m: Meter): Meter = new Meter(value + m.value)
  def toKm: Kilometer = new Kilometer(value / 1000)

class Kilometer(val value: Double) extends AnyVal:
  def toMeters: Meter = new Meter(value * 1000)

// Usage:
val distance = Meter(500) + Meter(300)
val inKm = distance.toKm  // No runtime overhead
```

Characteristics:

- Extends `AnyVal`
- No runtime allocation overhead
- Type safety without performance cost

Class Design Principles

- **Favor immutability**:

```
// Prefer
case class Point(x: Double, y: Double)

// Over
class MutablePoint(var x: Double, var y: Double)
```

- **Small, focused classes**:
  - Single Responsibility Principle
  - 50-100 lines max
- **Document invariants**:

```
class NonEmptyList[A](val head: A, val tail: List[
    A]):
  require(tail != null, "Tail cannot be null")
```

Trait Design Guidelines

- **Keep traits focused**:

```
// Good
trait Logging
trait Authentication
trait DatabaseAccess

// Bad
trait ControllerUtilities  // Too vague
```

- **Use sealed traits for closed hierarchies**:

```
sealed trait Response
case class Success(data: String) extends Response
case class Failure(error: String) extends Response
```

- **Linearization matters**:
  - Traits are stacked last-to-first
  - Put fundamental traits first

Object Usage Patterns

- **For type class instances**:

```
object JsonWriters:
  given JsonWriter[Int] = ...
  given JsonWriter[String] = ...
```

- **As modules**:

```
object DatabaseModule:
  def connect(config: Config) = ...
  def query(sql: String) = ...
```

- **For entry points**:

```
@main def runApp(): Unit =
  println("Application started")
```

Key Takeaways

- **Classes** are templates for objects with state and behavior
- **Objects** provide singleton instances and utilities
- **Traits** enable flexible composition of behavior
- Scala 3 adds:
  - Trait parameters
  - Open classes
  - Improved syntax
- Combine these features for clean, modular designs

Further Learning

- Official Scala 3 Documentation
- "Programming in Scala, 5th Edition" (Odersky et al.)
- "Scala with Cats" (Noel Welsh and Dave Gurnell)
- Scala Exercises (https://www.scala-exercises.org/)
- Contribute to open-source Scala projects