Case Classes

Quick Intro To Case Classes

The case class in scala is the most relevant feature in scala for handling data effectively.

```
final case class Employee(firstName: String, lastName: String, ssn: String)
```

The **final** qualifier makes sense because extending a case class may lead to inconsistencies and performance issues as well (https://gist.github.com/chaotic3quilibrium/58e78a2e21ce43bfe0042bbfbb93e7dc)

A case class provides a swiss knife of features to make your life easier when handling data.

Covering The Key Features

Here is the list of the most popular features provided by the case class:

- A convenient toString() method that will display all it's field contents
- A compare by-field-values, not be reference
- A copy method for handling immutable data
- Pattern matching for field extraction (perhaps better avoided)

Convenience toString Method

With case classes, the toString method is invoked when you need to evaluate any object to a string eg

```
final case class Employee(firstName: String, lastName: String, ssn: String)
class EmployeeClass(firstName: String, lastName: String, ssn: String)

val employee = Employee("john", "wick", "111-11-1111")

val employeeClass = new EmployeeClass("john", "wick", "111-11-1111")

println(employee)
// Cool
```

```
// Employee(john,wick,111–11–1111)

println(employeeClass)
// Not cool. Prints a representation of the reference for this object
// Main$EmployeeClass$1@3f3afe78
```

Equality By Structure Not By Reference

With case classes, you can compare objects by their structure not by reference (default when using plain classes). Here is an illustration on how it works vs plan classes:

```
final case class Employee(firstName: String, lastName: String, ssn: String)
class EmployeeClass(firstName: String, lastName: String, ssn: String)



val employee1 = Employee("john", "wick", "111–11–1111")
val employee2 = Employee("john", "wick", "111–11–1111")
val employee3 = Employee("robert", "mccall", "222–222–2222")
println(employee1 == employee2)
// true
println(employee1 == employee3)
// false

val employeeClass1 = new EmployeeClass("john", "wick", "111–11–1111")
val employeeClass2 = new EmployeeClass("john", "wick", "111–11–1111")
println(employeeClass1 == employeeClass2
// false
```

Built-In copy Method

Case classes are immutable by default. This means that modifying fields is not possible. However, you can copy-modify case classes. eg:

```
val employeeWithModifiedLastName = employee1.copy(firstName = "Jon")
println(employeeWithModifiedLastName)
// Employee(Jon,wick,111-11-1111)
println(employee1)
// FYI, employee1 is not changed
// Employee(john,wick,111-11-1111)
```

Note: cases classes may not be mutable by default, but they can be mutable eg:

```
final case class MutableEmployee(var firstName: String, var lastName:
  String, var ssn: String)

val mutableEmployee = MutableEmployee("gravik", "skrull", "333-33-3333")
println(mutableEmployee)
// MutableEmployee(gravik,skrull,333-33-3333)

mutableEmployee.firstName = "talos"
println(mutableEmployee)
// MutableEmployee(talos,skrull,333-33-3333)
```

Using Pattern Matching For Field Extraction

You can extract the fields of a case class by using pattern matching eg:

```
employee1 match {
  case Employee(firstName, lastName, ssn) =>
    println(s"Name is $firstName $lastName and ssn is $ssn")
  // Name is john wick and ssn is 111-11-1111
}

// No need to define unused fields
employee1 match {
  case Employee(firstName, _, _) =>
    println(s"First name is $firstName")
  // First name is john
}
```

The reason I discourage pattern match extractions for scala case classes is that correct extractions depends on the correct order of the fields. Also, adding an extra field to the case class will cause a compile error. Here is an example where the wrong order creates a bug:

```
// wrong order!
employee1 match {
  case Employee(lastName, firstName, ssn) =>
println(s"First name is $firstName, last name is $lastName and ssn is $ssn")
  // First name is wick, last name is john and ssn is 111-11-1111
}

final case class WideCaseClass(name: String, s1: String, s2: String, s3:
String,
                               r1: String, r2: String, r3: String, t1: String, t2: String,
t3: String)

val wideClass: WideCaseClass = WideCaseClass("a", "b", "c", "d", "e", "f", "g", "h", "i", "j")
wideClass match
case WideCaseClass(_, _, _, _, _, r1, _, _, _, _) => println(r1)
// You've got it wrong by one position! Too bad!!!
// Type safety won't help you here.
```

In this page we did the basic thing: introduced case classes and it's basic features: The toString method, the equality by structure, the copy method and my least favorite field extraction by pattern matching.

Separation Of Concerns: Case Class Access Scope, Validation, And Derived Fields

It may not be intuitive why scope and validation are together at this moment. Let me explain why these two topics should go together.

Basic Case Class With Validation

The most basic form of case class validation looks like the following. It works well but may not be ideal.

```
final case class EmployeeValidated(firstName: String, lastName: String, ssn: String):
  require(firstName.nonEmpty)
  require(lastName.nonEmpty)
  require(ssn.nonEmpty)

// Runs ok
val employee1: EmployeeValidated = EmployeeValidated("John", "Wick", "111-11-1111")

// Runtime exception thrown due to "require(lastName.nonEmpty)" validation above
val employee2: EmployeeValidated = EmployeeValidated("Michael", "", "222-22-2222")
```

Separating Validation And Case Classes

We typically want the case class to just be a placeholder of data. Mixing up validation, calculations and data in the same place will make case classes larger and bloated. For this purpose we want to use companion objects eg

```
final case class Employee(firstName: String, lastName: String, ssn: String)

object Employee:
  val ssnToFullName: Map[String, (String, String)]  = Map(
    "111-11-1111" -> ("John", "Wick"),
    "222-22-2222" -> ("Michael", "Bubble")
  )
  val fullNameToSsn: Map[(String, String), String]  =  ssnToFullName.map {case (k,v) =>
(v,k)}

  def fromAllFields(firstName: String, lastName: String, ssn: String): Employee =
    require(firstName.nonEmpty)
    require(lastName.nonEmpty)
    require(ssn.nonEmpty)
    Employee(firstName, lastName, ssn)

  def fromSsn(ssn: String): Employee =
    require(ssn.nonEmpty)
    val (firstName: String, lastName: String) = ssnToFullName(ssn)
    Employee(firstName, lastName, ssn)

  def fromFullName(firstName: String, lastName: String): Employee =
    require(firstName.nonEmpty)
    require(lastName.nonEmpty)
    val ssn: String = fullNameToSsn((firstName, lastName))
    Employee(firstName, lastName, ssn)

// The following 3 println() should print the same data
val employee1FromBuilder = Employee.fromSsn("111-11-1111")
println(employee1FromBuilder)
val employee2FromBuilder = Employee.fromAllFields("John", "Wick", "111-11-1111")
println(employee2FromBuilder)
val employee3FromBuilder = Employee.fromFullName("John", "Wick")
println(employee3FromBuilder)
```

Restricting Access To Case Class Constructor

Now the problem with the code above is that anyone outside this file can directly build Employee directly without validation. Ideally, we want to restrict this to guarantee data integrity. A way to address this issue is to add a **private** qualifier to the case class right before the parenthesis eg

```
final case class EmployeePrivate private (firstName: String, lastName: String, ssn: String)

object EmployeePrivate:
  val ssnToFullName: Map[String, (String, String)]  = Map(
    "111-11-1111" -> ("John", "Wick"),
    "222-22-2222" -> ("Michael", "Bubble")
  )
  val fullNameToSsn: Map[(String, String), String]  =  ssnToFullName.map {case (k,v) =>
(v,k)}

  def fromAllFields(firstName: String, lastName: String, ssn: String): EmployeePrivate =
    require(firstName.nonEmpty)
    require(lastName.nonEmpty)
    require(ssn.nonEmpty)
    EmployeePrivate(firstName, lastName, ssn)

  def fromSsn(ssn: String): EmployeePrivate =
    require(ssn.nonEmpty)
    val (firstName: String, lastName: String) = ssnToFullName(ssn)
    EmployeePrivate(firstName, lastName, ssn)

  def fromFullName(firstName: String, lastName: String): Employee =
    require(firstName.nonEmpty)
    require(lastName.nonEmpty)
    val ssn: String = fullNameToSsn((firstName, lastName))
    Employee(firstName, lastName, ssn)

// The following 3 println() should print the same data
val employeePrivate1FromBuilder = EmployeePrivate.fromSsn("111-11-1111")
println(employee1FromBuilder)
val employeePrivate2FromBuilder = EmployeePrivate.fromAllFields("John", "Wick", "111-11-1111")
println(employee2FromBuilder)
```

```
val employeePrivate3FromBuilder = EmployeePrivate.fromFullName("John", "Wick")
println(employeePrivate3FromBuilder)
```

Now you won't be able to instantiate EmployeePrivate directly. You will only be able to do it through the companion object or any other method in the same file.

```
// Code from a separate file
import CaseClassAccessScopeAndValidation.EmployeePrivate

// The following will create a compile error
val employee = EmployeePrivate("John", "Wick", "111-11-1111")
// method apply cannot be accessed as a member of
CaseClassAccessScopeAndValidation.EmployeePrivate.
// type from module class CaseClassAccessScopeAndValidation_AccessAtempt$.

// But the following will work just fine
val employee2 = EmployeePrivate.fromAllFields("John", "Wick", "111-11-1111")
```

Note: be aware that the case class convenience methods, namely copy, will not be available when you add the **private** qualifier eg:

```
// Code from same separate file as code sample above

val employee2 = EmployeePrivate.fromAllFields("John", "Wick", "111-11-1111")

// Following code will generate a compile error
val employee3 = employee2.copy(firstName = "hernan")
// from module class CaseClassAccessScopeAndValidation_AccessAtempt$
// method copy cannot be accessed as a member of
// (CaseClassAccessScopeAndValidation_AccessAtempt.employee :
CaseClassAccessScopeAndValidation.EmployeePrivate )
```

In that case, you will have to implement the copy command in a method inside the companion object. Keep in mind the companion object has access to all the private methods of the case class and vice-versa.

Keep Derived Fields Outside The Case Class

We still want to stick to the idea of only keeping data in the case class and keep any logic outside

of it. Same thing applies to derived fields (fields that are a calculation of other fields in the case class).

There are at least 2 ways we can do this. **Method 1**: Create a class that extends the case class and a trait containing the derived field. And **Method 2**: Use scala **extension**.

```
case class NonFinalEmployee(firstName: String, lastName: String, ssn: String)

// Method 1 for adding derived fields
// Use a trait and extend Employee case class with trait containing the derived field
trait FullNameDerived:
  self: NonFinalEmployee =>
    def fullNameDerived1 =
      s"$firstName $lastName"

class EmployeeData(firstName: String, lastName: String, ssn: String)
  extends NonFinalEmployee(firstName, lastName, ssn) with FullNameDerived

val employeeData: EmployeeData = new EmployeeData("John", "Wick", "111-11-1111")
println(employeeData.fullNameDerived1)

// Method 2 for adding derived fields
// Use extension method and extend Employee with derived field instead
extension (c: Employee)
  def fullNameDerived2: String=
    s"${c.firstName} ${c.lastName}"

val employee = Employee ("John", "Wick", "111-11-1111")
println(employee.fullNameDerived2)
```

Note: The author of this article prefers **method 2** for looking cleaner and being more extensible. Besides, we can still keep the case class **final** as well.