Case Classes and Pattern Matching

Jacek Wasilewski

Case Classes (1)

- Quite often we create classes that
  - simple, just take parameters and store them
  - do not change after creation
- In Scala – **case classes**.
- Syntax

```
case class Person(firstName: String, lastName: String)
```

Case Classes (2)

- Constructors parameters become public members
- Methods `toString()`, `equals()` and `hashCode()` are implemented
- Companion object is created with
  - `apply()` method that mimics the constructor
  - `unapply()` extractor
- Object can be created without the `new` keyword
  `Person("Jacek", "Wasilewski")`

Case Classes (3)

- ```
  scala> case class Person(firstName: String, lastName:
  String)
  defined class Person
  ```
- ```
  scala> val p = Person("Jacek", "Wasilewski")
  p: Person = Person(Jacek,Wasilewski)
  ```
- ```
  scala> p.firstName
  res0: String = Jacek
  ```
- ```
  scala> p.lastName
  res1: String = Wasilewski
  ```

Case Classes (4)

- scala> p.toString()
  res2: String = Person(Jacek,Wasilewski)
- scala> p.equals(Person("Jacek", "Wasilewski"))
  res3: Boolean = true
- scala> p.hashCode
  res4: Int = 1857982383
- scala> Person.apply("Jacek","Wasilewski")
  res5: Person = Person(Jacek,Wasilewski)
- scala> Person.unapply(p)
  res6: Option[(String, String)] = Some((Jacek,Wasilewski))

Case Classes (5)

- Case class is converted to a typical class
- Parameter list has to be defined – even empty

```
case class Test()
case class Test
```

- Case class can inherit from other class:

```
abstract class A
case class B() extends A
```

- Class can extend case class:

```
case class X()
class Y extends X()
```

- Case class **can not** extend other case class.

```
case class Z() extends X()
```

Case Classes (6)

- Case classes are very useful in **pattern matching**.
- Let's take a look at the accompanying object that has been created:
```
apply(firstName, lastName): Person
unapply(p: Person): Option[(String, String)]
```
- `apply` method creates a new object.
- `unapply` decomposes an object into a structure that holds constructor's parameters.

Pattern Matching (1)

- Built-in general pattern matching mechanism
- Allows to match any sort of data with a first match policy
- Like `switch` but more powerful
- `match` and `case` keywords
- More sophisticated matching – on classes (using case classes)

Pattern Matching (2)

- Patterns can be constructed using
  - constructors – `Number`, `Sum`,
  - variables – `n`, `e1`, `e2`,
  - wildcard patterns - `_`,
  - constants – `1`, `true`.
- Syntax

```
def matchTest(x: Int): String = x match {
  case 1 => "one"
  case 2 => "two"
  case _ => "something"
}
```

Pattern Matching (3)

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

def matchPerson(p: Person):String = p match {
  case Person("Alice", 25) => "Hi Alice!"
  case Person("Bob", 32) => "Hi Bob!"
  case Person("Alice", age) => "Hi Alice, are you " + age + "y old?"
  case Person(name, 32) => "Hi " + name + ", I know you are 32!"
  case Person(name, age) => "Hi " + name + ", you are " + age + "y old"
}
```

```
scala> matchPerson(Person("Alice", 25))
res7: String = Hi Alice!

scala> matchPerson(Person("Bob", 32))
res8: String = Hi Bob!

scala> matchPerson(Person("Bob", 25))
res9: String = Hi Bob, you are 25y old
```

Example (1)

- Problem: we would like to implement a set of classes to represent expressions of different forms.
- Sample expression is: 3 + ((5 + 3) + 4).
- Also we would like to evaluate these expressions.
- In this case, we need to model: `Expression`, `Number` and `Sum`.

Example (2)

```
abstract class Expression {
    def eval: Int
}

class Number(n: Int) extends Expression {
    def eval: Int = n
}

class Sum(e1: Expression, e2: Expression) extends Expression
{
    def eval: Int = e1.eval + e2.eval
}
```

Example (3)

- With case classes

```
case class Number(n: Int) extends Expression {
    def eval: Int = n
}

case class Sum(e1: Expression, e2: Expression) extends
Expression {
    def eval: Int = e1.eval + e2.eval
}
```

- Nice, object-oriented way of decomposing the problem
- Not the only solution

Example (4)

- Let's say we would like to control the evaluation in one place
  - so rules for evaluation sum etc. would be in one block
- Let's match like persons before

```
abstract class Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr

def eval(e: Expr): Int = e match {
    case Number(n) => n
    case Sum(e1, e2) => eval(e1) + eval(e2)
}
```

Example (5)

- Adding new expression

```
case class Prod(e1: Expr, e2: Expr) extends Expr
def eval(e: Expr): Int = e match {
    ...
    case Prod(e1, e2) => eval(e1) * eval(e2)
}
```

Example (6)

- Cases can be more complex

```
def eval(e: Expr): Int = e match {
  ...
  case Sum(Prod(e1, e2), Prod(e3, e4) => {
    if (e1 == e3 && e2 == e4)
      2 * eval(e1) * eval(e2)
    else
      eval(e1) * eval(e2) + eval(e3) * eval(e4)
  }
}
```

Lists and Pattern Matching (1)

- Lists work well with pattern matching
- Reminder: lists have recursive structure – head and tail
- List can be represented as **`head :: tail`**
  - and decomposed in the same way
- Code:

```
def removeHead(list: List[Int]): List[Int] = list match {
    case List() => Nil
    case List(e) => Nil
    case head :: tail => tail
}
```

Lists and Pattern Matching (2)

```
val list: List[Int] = List(1, 2, 3, 4, 5)

def reverse(list: List[Int]): List[Int] = list match {
  case head :: tail => {
    reverse(tail) ::: List(head)
  }
  case _ => list
}


scala> reverse(list)
res30: List[Int] = List(5, 4, 3, 2, 1)
```

References

- “Scala By Example”, Martin Odersky, EPFL
- “Functional Programming Principles in Scala”, Martin Odersky, Coursera
- http://alvinalexander.com/scala/scala-class-examples-constructors-case-classes-parameters
- https://twitter.github.io/scala_school/basics2.html#caseclass
- http://docs.scala-lang.org/tutorials/tour/case-classes.html