TOUR OF SCALA

EXTRACTOR OBJECTS

An extractor object is an object with an `unapply` method. Whereas the `apply` method is like a constructor which takes arguments and creates an object, the `unapply` takes an object and tries to give back the arguments. This is most often used in pattern matching and partial functions.

```scala
import scala.util.Random

object CustomerID {

  def apply(name: String) = s"$name--${Random.nextLong}"

  def unapply(customerID: String): Option[String] = {
    val stringArray: Array[String] = customerID.split("--")
    if (stringArray.tail.nonEmpty) Some(stringArray.head) else None
  }
}

val customer1ID = CustomerID("Sukyoung")  // Sukyoung--23098234908
customer1ID match {
  case CustomerID(name) => println(name)  // prints Sukyoung
  case _ => println("Could not extract a CustomerID")
}
```

The `apply` method creates a `CustomerID` string from a `name`. The `unapply` does the inverse to get the `name` back. When we call `CustomerID("Sukyoung")`, this is shorthand syntax for calling `CustomerID.apply("Sukyoung")`. When we call `case CustomerID(name) => println(name)`, we're calling the unapply method with `CustomerID.unapply(customer1ID)`.

Since a value definition can use a pattern to introduce a new variable, an extractor can be used to initialize the variable, where the unapply method supplies the value.

```scala
val customer2ID = CustomerID("Nico")
val CustomerID(name) = customer2ID
println(name)  // prints Nico
```

This is equivalent to `val name = CustomerID.unapply(customer2ID).get`.

```scala
val CustomerID(name2) = "--asdfasdfasdf"
```

If there is no match, a `scala.MatchError` is thrown:

```scala
val CustomerID(name3) = "-asdfasdfasdf"
```

The return type of an `unapply` should be chosen as follows:

- If it is just a test, return a `Boolean`. For instance `case even()`.
- If it returns a single sub-value of type T, return an `Option[T]`.
- If you want to return several sub-values `T1,...,Tn`, group them in an optional tuple `Option[(T1,...,Tn)]`

- If you want to return several sub-values `T1,...,Tn`, group them in an optional tuple `Option[(T1,...,Tn)]`.

Sometimes, the number of values to extract isn't fixed and we would like to return an arbitrary number of values, depending on the input. For this use case, you can define extractors with an `unapplySeq` method which returns an `Option[Seq[T]]`. Common examples of these patterns include deconstructing a `List` using `case List(x, y, z) =>` and decomposing a `String` using a regular expression `Regex`, such as `case r(name, remainingFields @ _*) =>`.

← **previous** **next** →

Contributors to this page:

ckipp01 mlachkar vegerot ashawley som-snytt luiru72 Gommorach
miller-time jbalintbiro heathermiller