Option

Effective Programming in Scala

Option

The special case of a collection that has at most one element is named `Option`.

Like the collections, the type `Option` is parameterized by the type of element: `Option[Int]` is the type of an optional integer value, `Option[Boolean]` is the type of an optional boolean value, etc.

An optional value can either be `None` (absence of value), or `Some(value)`.

Optional values have similar operations as collections (`map`, `filter`, `flatMap`, `foldLeft`, `isEmpty`, `contains`, etc.).

Address Book Example: Optional Email

Let us revisit the `Contact` data type and make the `email` field optional:

```
case class Contact(
  name: String,
  maybeEmail: Option[String],
  phoneNumbers: List[String]
)

val alice = Contact("Alice", Some("alice@sca.la"), List())
val bob   = Contact("Bob", None, List("+41787829420"))
```

Address Book Example: Optional Email

Let us revisit the `Contact` data type and make the `email` field optional:

```
case class Contact(
  name: String,
  maybeEmail: Option[String],
  phoneNumbers: List[String]
)

val alice = Contact("Alice", Some("alice@sca.la"), List())
val bob   = Contact("Bob", None, List("+41787829420"))
```

- It is common practice to prefix identifiers of optional values with “maybe” as in `maybeEmail`

Pattern Matching on Option

```
def hasScaDotLaEmail(contact: Contact): Boolean =
  contact.maybeEmail match
    case Some(email) => email.endsWith(”@sca.la”)
    case None        => false
```

Other Operations on Option

The operation `map` transforms the element in the option with the given function. The operation `getOrElse` returns the optional value, if it is defined, or falls back to a given value.

```
def emailLength(contact: Contact): Int =
  contact.maybeEmail
    .map(email => email.size)
    .getOrElse(0)
```

The operation `zip` combines two optional values into a single optional value containing a pair. The resulting value is defined only if the two original optional values were defined.

```
val maybeAliceAndBobEmails: Option[(String, String)] =
  alice.maybeEmail.zip(bob.maybeEmail)
```

Collection Operations Returning Optional Results

We have already seen that the operation `find` on collections returns `None` in case no elements were found.

Another example is the operation `headOption` on `List` that optionally returns the first element of the list if the list is not empty.

```scala
def sendNotification(contact: Contact, message: String): Unit =
  contact.phoneNumbers.headOption match
    case Some(number) => sendSms(number, message)
    case None =>
      contact.maybeEmail match
        case Some(email) => sendEmail(email, message)
        case None        => ()
```

`null`

In some programming languages, the value `null` (or equivalent) is used to model optional values. However, it is now largely admitted that this is bad practice:

`https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare`

In Scala, there is a `null` value, but it is here mostly for interoperability with Java. We recommend modeling optional values with the type `Option` instead.

Summary

The type `Option` models optional values.

A value of type `Option[A]` can either be `None`, or `Some(a: A)`.

Some collection operations return optional values (e.g., `find`, `headOption`).