Open Classes

An `open` modifier on a class signals that the class is planned for extensions. Example:

```
// File Writer.scala
package p

open class Writer[T]:

  /** Sends to stdout, can be overridden */
  def send(x: T) = println(x)

  /** Sends all arguments using `send` */
  def sendAll(xs: T*) = xs.foreach(send)
end Writer

// File EncryptedWriter.scala
package p

class EncryptedWriter[T: Encryptable] extends Writer[T]:
  override def send(x: T) = super.send(encrypt(x))
```

An open class typically comes with some documentation that describes the internal calling patterns between methods of the class as well as hooks that can be overridden. We call this the *extension contract* of the class. It is different from the *external contract* between a class and its users.

Classes that are not open can still be extended, but only if at least one of two alternative conditions is met:

- The extending class is in the same source file as the extended class. In this case, the extension is usually an internal implementation matter.
- The language feature `adhocExtensions` is enabled for the extending class. This is typically enabled by an import clause in the source file of the extension:

```
import scala.language.adhocExtensions
```

Alternatively, the feature can be enabled by the compiler option `-language:adhocExtensions`. If the feature is not enabled, the compiler will issue a "feature" warning. For instance, if the `open` modifier on class `Writer` is dropped, compiling `EncryptedWriter` would produce a warning:

```
-- Feature Warning: EncryptedWriter.scala:6:14 ----
  |class EncryptedWriter[T: Encryptable] extends Writer[T]
  |                                              ^
  |Unless class Writer is declared 'open', its extension
  | in a separate file should be enabled
  |by adding the import clause 'import scala.language.adhocExtensions'
  |or by setting the compiler option -language:adhocExtensions.
```

Motivation

When writing a class, there are three possible expectations of extensibility:

1. The class is intended to allow extensions. This means one should expect a carefully worked out and documented extension contract for the class.
2. Extensions of the class are forbidden, for instance to make correctness or security guarantees.
3. There is no firm decision either way. The class is not *a priori* intended for extensions, but if others find it useful to extend on an *ad-hoc* basis, let them go ahead. However, they are on their own in this case. There is no documented extension contract, and future versions of the class might break the extensions (by rearranging internal call patterns, for instance).

The three cases are clearly distinguished by using `open` for (1), `final` for (2) and no modifier for (3).

It is good practice to avoid *ad-hoc* extensions in a code base, since they tend to lead to fragile systems that are hard to evolve. But there are still some situations where these extensions are useful: for instance, to mock classes in tests, or to apply temporary patches that add features or fix bugs in library classes. That's why *ad-hoc* extensions are permitted, but only if there is an explicit opt-in via a language feature import.

Details

- `open` is a soft modifier. It is treated as a normal identifier unless it is in modifier position.
- An `open` class cannot be `final` or `sealed`.
- Traits or `abstract` classes are always `open`, so `open` is redundant for them.

Relationship with `sealed`

A class that is neither `abstract` nor `open` is similar to a `sealed` class: it can still be extended, but only in the same source file. The difference is what happens if an extension of the class is attempted in another source file. For a `sealed` class, this is an error, whereas for a simple non-open class, this is still permitted provided the `adhocExtensions` feature is enabled, and it gives a warning otherwise.

Migration

`open` is a new modifier in Scala 3. To allow cross compilation between Scala 2.13 and Scala 3.0 without warnings, the feature warning for ad-hoc extensions is produced only under `-source future`. It will be produced by default from Scala 3.1 on.

‹ Opaqu... Param... ›