Extension Methods

Effective Programming in Scala

Motivation

Since opaque types have no methods, you have to define auxiliary methods to work with opaque types.

```
object UserID:
  opaque type UserID = Long
  def value(userID: UserID): Long = userID
```

For instance, to retrieve the underlying `Long` value of a user ID you would write:

```
UserID.value(userID)
```

Motivation

Since opaque types have no methods, you have to define auxiliary methods to work with opaque types.

```
object UserID:
  opaque type UserID = Long
  def value(userID: UserID): Long = userID
```

For instance, to retrieve the underlying `Long` value of a user ID you would write:

```
UserID.value(userID)
```

Instead, it would be more practical to just write:

```
userID.value
```

Extension Methods

We can define a method `value` to the type `UserID` by writing an **extension method**.

```
object UserID:
  opaque type UserID = Long
  extension (userID: UserID)
    def value: Long = userID

import UserID.UserID

def findUser(userID: UserID): Option[User] =
  ...
  ... userID.value
  ...
```

Extension Methods (2)

Extension methods are not specific to opaque types.

In fact, you can use this mechanism to enrich any type with additional methods:

```
extension (n: Int)
  def isZero: Boolean = n == 0
  def ** (e: Int): Int = List.fill(e)(n).product

42.isZero // : Boolean = false
0.isZero  // : Boolean = true
5 ** 3    // : Int     = 125
```

Translation of Extension Methods

When you define an extension method like the following:

```
extension (n: Int) def ** (e: Int): Int = ...
```

The compiler translates extension methods to regular methods, similar to:

```
def ** (n: Int)(e: Int): Int = ...
```

As a consequence, extension methods can also be called with the following syntax:

```
**(2)(4) // : Int = 16
```

2 ** 4

Applicability of Extension Methods

Since the type `UserID` has no actual method `value`, and the type `Int` has no actual method `isZero`, how does the compiler resolve calls to extension methods?

When you call a method `m` on an expression `e` of type `E`, like in `e.m`, and that the type `E` has no method `m`, the compiler tries to rewrite the call into `m(e)`.

This is what happens when we write `42.isZero`. This expression does not type check as is. The compiler tries `isZero(42)`, which does type check.

Note that `isZero(42)` does type check only if the extension method `isZero` is *visible* at the point where the call is performed.

Visibility of Extension Methods

```
trait IntExtensions:
  extension (n: Int) def isZero: Boolean = n == 0

  0.isZero // OK, isZero is defined in the enclosing scope

object IntExtensions extends IntExtensions:
  1.isZero // OK, isZero is inherited from the trait IntExtensions

2.isZero // Error: value isZero is not a member of Int
isZero(2)
import IntExtensions.isZero

3.isZero // OK, isZero is imported
```

Extension Methods vs Methods

Since the compiler looks for extensions only when the method that is called is missing, extensions can only add new members, not override existing ones.

Extensions cannot refer to other class members via `this`.

Applicability of Extension Methods (2)

Consider again the following program:

```
object UserID:
  opaque type UserID = Long
  extension (userID: UserID) def value: Long = userID

def findUser(userID: UserID.UserID): Option[User] =
  ...
  ... userID.value      // value(userID)
  ...
```

Here, the extension method `value` is not visible at the point where the call `userID.value` is performed, in the method `findUser`.

Why don't we need to `import UserID.value`?

Applicability of Extension Methods (3)

In the case of `userID.value`, we don't need to import the extension method because there is an additional rule applied by the compiler.

Since the type `UserID` is an opaque type, the compiler also looks for extension methods in its scope of definition (in this case, the object `UserID`), and it finds the method `UserID.value`.

(There are additional rules that will be detailed later in the course.)

Summary

Extension methods let you introduce new methods to existing types.

The compiler allows you to apply extension methods if they are visible (defined in an enclosing scope, inherited, or imported) at the point of application.

In the case of opaque types, the compiler also looks for extension methods in the scope of definition of the opaque type.