Opaque Type Aliases

Edit this page on GitHub

Opaque types aliases provide type abstraction without any overhead. Example:

```
object MyMath:

  opaque type Logarithm = Double

  object Logarithm:

    // These are the two ways to lift to the Logarithm type

    def apply(d: Double): Logarithm = math.log(d)

    def safe(d: Double): Option[Logarithm] =
      if d > 0.0 then Some(math.log(d)) else None

  end Logarithm

  // Extension methods define opaque types' public APIs
  extension (x: Logarithm)
    def toDouble: Double = math.exp(x)
    def + (y: Logarithm): Logarithm = Logarithm(math.exp(x) + math.exp(y))
    def * (y: Logarithm): Logarithm = x + y

end MyMath
```

This introduces `Logarithm` as a new abstract type, which is implemented as `Double`. The fact that `Logarithm` is the same as `Double` is only known in the scope where `Logarithm` is defined, which in the above example corresponds to the object `MyMath`. Or in other words, within the scope, it is treated as a type alias, but this is opaque to the outside world where, in consequence, `Logarithm` is seen as an abstract type that has nothing to do with `Double`.

The public API of `Logarithm` consists of the `apply` and `safe` methods defined in the companion object. They convert from `Double`s to `Logarithm` values. Moreover, an operation `toDouble` that converts the other way, and operations `+` and `*` are defined as extension methods on `Logarithm` values. The following operations would be valid because they use functionality implemented in the `MyMath` object.

```
import MyMath.Logarithm

val l = Logarithm(1.0)
val l2 = Logarithm(2.0)
val l3 = l * l2
val l4 = l + l2
```

But the following operations would lead to type errors:

```
val d: Double = l       // error: found: Logarithm, required: Double
val l2: Logarithm = 1.0 // error: found: Double, required: Logarithm
l * 2                   // error: found: Int(2), required: Logarithm
l / l2                  // error: `/` is not a member of Logarithm
```

Bounds For Opaque Type Aliases

Opaque type aliases can also come with bounds. Example:

```
object Access:

  opaque type Permissions = Int
  opaque type PermissionChoice = Int
  opaque type Permission <: Permissions & PermissionChoice = Int

  extension (x: PermissionChoice)
    def | (y: PermissionChoice): PermissionChoice = x | y
  extension (x: Permissions)
    def & (y: Permissions): Permissions = x | y
  extension (granted: Permissions)
    def is(required: Permissions) = (granted & required) == required
    def isOneOf(required: PermissionChoice) = (granted & required) != 0

  val NoPermission: Permission = 0
  val Read: Permission = 1
  val Write: Permission = 2
  val ReadWrite: Permissions = Read | Write
  val ReadOrWrite: PermissionChoice = Read | Write

end Access
```

The `Access` object defines three opaque type aliases:

- `Permission`, representing a single permission,
- `Permissions`, representing a set of permissions with the meaning "all of these permissions granted",
- `PermissionChoice`, representing a set of permissions with the meaning "at least one of these permissions granted".

Outside the `Access` object, values of type `Permissions` may be combined using the `&` operator, where `x & y` means "all permissions in `x` *and* in `y` granted". Values of type `PermissionChoice` may be combined using the `|` operator, where `x | y` means "a permission in `x` *or* in `y` granted".

Note that inside the `Access` object, the `&` and `|` operators always resolve to the corresponding methods of `Int`, because members always take precedence over extension methods. For that reason, the `|` extension method in `Access` does not cause infinite recursion.

In particular, the definition of `ReadWrite` must use `|`, the bitwise operator for `Int`, even though client code outside `Access` would use `&`, the extension method on `Permissions`. The internal representations of `ReadWrite` and `ReadOrWrite` are identical, but this is not visible to the client, which is interested only in the semantics of `Permissions`, as in the example below.

All three opaque type aliases have the same underlying representation type `Int`. The `Permission` type has an upper bound `Permissions & PermissionChoice`. This makes it known outside the `Access` object that `Permission` is a subtype of the other two types. Hence, the following usage scenario type-checks.

```
object User:
  import Access.*

  case class Item(rights: Permissions)
  extension (item: Item)
    def +(other: Item): Item = Item(item.rights & other.rights)

  val roItem = Item(Read)  // OK, since Permission <: Permissions
  val woItem = Item(Write)
  val rwItem = Item(ReadWrite)
  val noItem = Item(NoPermission)

  assert(!roItem.rights.is(ReadWrite))
  assert(roItem.rights.isOneOf(ReadOrWrite))
```

```
  assert(rwItem.rights.is(ReadWrite))
  assert(rwItem.rights.isOneOf(ReadOrWrite))

  assert(!noItem.rights.is(ReadWrite))
  assert(!noItem.rights.isOneOf(ReadOrWrite))

  assert((roItem + woItem).rights.is(ReadWrite))
end User
```

On the other hand, the call `roItem.rights.isOneOf(ReadWrite)` would give a type error:

```
assert(roItem.rights.isOneOf(ReadWrite))
                             ^^^^^^^^^
                             Found:    (Access.ReadWrite : Access.Permissions
                             Required: Access.PermissionChoice
```

`Permissions` and `PermissionChoice` are different, unrelated types outside `Access`.

Opaque Type Members on Classes

While typically, opaque types are used together with objects to hide implementation details of a module, they can also be used with classes.

For example, we can redefine the above example of Logarithms as a class.

```
class Logarithms:

  opaque type Logarithm = Double

  def apply(d: Double): Logarithm = math.log(d)

  def safe(d: Double): Option[Logarithm] =
    if d > 0.0 then Some(math.log(d)) else None

  def mul(x: Logarithm, y: Logarithm) = x + y
```

Opaque type members of different instances are treated as different:

```
val l1 = new Logarithms
val l2 = new Logarithms
val x = l1(1.5)
val y = l1(2.6)
val z = l2(3.1)
l1.mul(x, y) // type checks
l1.mul(x, z) // error: found l2.Logarithm, required l1.Logarithm
```

In general, one can think of an opaque type as being only transparent in the scope of `private[this]`.

More details

‹ Export ... Opaqu... ›