



Opaque Types

Effective Programming in Scala

Motivation

Consider the following API for manipulating users and vehicles.

```
def findVehicle(vehicleID: Long): Option[Vehicle] = ...
```


```
def mistake(userID: Long): Unit =  
  findVehicle(userID) // What?!
```

Motivation

Consider the following API for manipulating users and vehicles.

```
def findVehicle(vehicleID: Long): Option[Vehicle] = ...
```

```
def mistake(userID: Long): Unit =  
  findVehicle(userID) // What?!
```



Problem: since both vehicle IDs and user IDs are modeled with type Long, we can mix them!

Encapsulating ID Types

```
case class UserID private (value: Long)
```

```
object UserID:
```

```
  def parse(string: String): Option[UserID] =  
    string.toLongOption.map(id => UserID(id))
```

Encapsulating ID Types

```
case class UserID private (value: Long)

object UserID:
  def parse(string: String): Option[UserID] =
    string.toLongOption.map(id => UserID(id))
```

- ▶ This code defines a *public* type UserID, but its constructor is *private* (it can be accessed from within the UserID companion object, though)
- ▶ The only way to construct a value of type UserID is to use the operation UserID.parse

Encapsulating ID Types (2)

```
def findVehicle(vehicleID: VehicleID): Option[Vehicle] = ...
```

```
def mistake(userID: UserID): Unit =  
  findVehicle(userID)
```

Encapsulating ID Types (2)

```
def findVehicle(vehicleID: VehicleID): Option[Vehicle] = ...
```

```
def mistake(userID: UserID): Unit =  
  findVehicle(userID)
```

^^^^^

Found: (userID : UserID)

Required: VehicleID

This nonsensical program is now rejected!

Encapsulating ID Types (2)

```
def findVehicle(vehicleID: VehicleID): Option[Vehicle] = ...
```

```
def mistake(userID: UserID): Unit =  
  findVehicle(userID)
```

^^^^^

Found: (userID : UserID)

Required: VehicleID

This nonsensical program is now rejected!

However, the UserID and VehicleID abstractions have a (small) cost: they instantiate a wrapper object.

Type Aliases

One way to remove the runtime overhead is to use a **type alias**:

```
type UserID = Long
```

```
object UserID:
```

```
  def parse(string: String): Option[UserID] =  
    string.toLongOption
```

//: Option[Long]

Type aliases incur no runtime costs (there is no class added), and can be interchangeably used with the type they are an alias to.

Type Aliases (2)

Type aliases are also handy to provide a shorthand name for a complex type expression:

```
type AgeAndUsers = (Int, Seq[User])
```

```
val users: Seq[User] = ...
```

```
val usersByAge: Iterable[AgeAndUsers] =  
  users.groupBy(user => user.age)
```

```
// equivalent to 'val usersByAge: Iterable[(Int, Seq[User])]'
```

“Transparent” Type Aliases

However, with simple type aliases, we are back to square one...

```
type UserID    = Long  
type VehicleID = Long
```

```
def findVehicle(vehicleID: VehicleID): Option[Vehicle] = ...
```

```
def mistake(userID: UserID): Unit =  
  findVehicle(userID) // What?!
```

Opaque Type Aliases

Opaque type aliases offer the best of both worlds:

```
object UserID:
  opaque type UserID = Long
object VehicleID
  opaque type VehicleID = Long
```

```
import UserID.UserID
import VehicleID.VehicleID
def findVehicle(vehicleID: VehicleID): Option[Vehicle] = ...
def mistake(userID: UserID): Unit =
  findVehicle(userID)
```

^^^^^

Found: (userID : UserID.UserID)

Required: VehicleID.VehicleID

Opaque Type Aliases

- ▶ Like type aliases, opaque type aliases incur no runtime overhead,
- ▶ Inside the scope of the alias definition, the alias is transparent:

```
object UserID:  
  opaque type UserID = Long  
  // Within the object ID, the type User can be used  
  // interchangeably with the type Long  
end UserID
```

- ▶ Outside its scope, the alias is “opaque”: it hides the type it is an alias to.

$UserID.UserID \neq Long$

Working with Opaque Types

By definition, there is nothing one can do with a value of type `UserID`

The object that defines the opaque type alias should also define methods that *produce* and *consume* opaque types:

```
object UserID:  
  opaque type UserID = Long  
  def parse(string: String): Option[UserID] = string.toLongOption  
  def value(userID: UserID): Long = userID
```

Summary

Type aliases let you give a name to a type expression.

Opaque type aliases encapsulate the type they are an alias to.

Opaque type aliases are “transparent” inside the scope they are defined, but they are “opaque” outside.

Define zero-cost abstractions with opaque types.