Class Hierarchies

Principles of Functional Programming

Abstract Classes

Consider the task of writing a class for sets of integers with the following operations.

```
abstract class IntSet {
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
}
```

`IntSet` is an *abstract class*.

Abstract classes can contain members which are missing an implementation (in our case, both `incl` and `contains`); these are called *abstract members*.

Consequently, no direct instances of an abstract class can be created, for instance an `IntSet()` call would be illegal.

Class Extensions

Let's consider implementing sets as binary trees.

There are two types of possible trees: a tree for the empty set, and a tree consisting of an integer and two sub-trees.

Here are their implementations:

```
class Empty() extends IntSet:
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = NonEmpty(x, Empty(), Empty())
end Empty
```

Class Extensions (2)

```
class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet:

  def contains(x: Int): Boolean =
    if x < elem then left.contains(x)
    else if x > elem then right.contains(x)
    else true

  def incl(x: Int): IntSet =
    if x < elem then NonEmpty(elem, left.incl(x), right)
    else if x > elem then NonEmpty(elem, left, right.incl(x))
    else this

end NonEmpty
```

Terminology

`Empty` and `NonEmpty` both *extend* the class `IntSet`.

This implies that the types `Empty` and `NonEmpty` *conform* to the type `IntSet`, i.e.

- an object of type `Empty` or `NonEmpty` can be used wherever an object of type `IntSet` is required.

Base Classes and Subclasses

`IntSet` is called the *superclass* of `Empty` and `NonEmpty`.

`Empty` and `NonEmpty` are *subclasses* of `IntSet`.

In Scala, any user-defined class extends another class.

If no superclass is given, the standard class `Object` in the Java package `java.lang` is assumed.

The direct or indirect superclasses of a class `C` are called *base classes* of `C`.

So, the base classes of `NonEmpty` are `IntSet` and `Object`.

Implementation and Overriding

The definitions of `contains` and `incl` in the classes `Empty` and `NonEmpty` *implement* the abstract functions in the base trait `IntSet`.

It is also possible to *redefine* an existing, non-abstract definition in a subclass by using `override`.

Example

```
abstract class Base:
  def foo = 1
  def bar: Int
```

```
class Sub extends Base:
  override def foo = 2
  def bar = 3
```

Object Definitions

In the `IntSet` example, one could argue that there is really only a single empty `IntSet`.

So it seems overkill to have the user create many instances of it.

We can express this case better with an *object definition*:

```
object Empty extends IntSet:
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = NonEmpty(x, Empty, Empty)
end Empty
```

This defines a *singleton object* named `Empty`.

No other `Empty` instances can be (or need to be) created.

Singleton objects are values, so `Empty` evaluates to itself.

Companion Objects

An object and a class can have the same name. This is possible since Scala has two global *namespaces*: one for types and one for values.

Classes live in the type namespace, whereas objects live in the term namespace.

If a class and object with the same name are given in the same sourcefile, we call them *companions*. Example:

```
class IntSet ...
object IntSet:
  def singleton(x: Int) = NonEmpty(x, Empty, Empty)
```

This defines a method to build sets with one element, which can be called as `IntSet.singleton(elem)`.

A companion object of a class plays a role similar to static class definitions in Java (which are absent in Scala).

Programs

So far we have executed all Scala code from the REPL or the worksheet.

But it is also possible to create standalone applications in Scala.

Each such application contains an object with a `main` method.

For instance, here is the "Hello World!" program in Scala.

```
object Hello:
  def main(args: Array[String]): Unit = println("hello world!")
```

Once this program is compiled, you can start it from the command line with

```
> scala Hello
```

Programs (2)

Writing `main` methods is similar to what Java does for programs.

Scala also has a more convenient way to do it.

A stand-alone application is alternatively a function that's annotated with `@main`, and that can take command line arguments as parameters:

```
@main def birthday(name: String, age: Int) =
  println(s"Happy birthday, $name! $age years old already!")
```

Once this function is compiled, you can start it from the command line with

```
> scala birthday Peter 11
Happy Birthday, Peter! 11 years old already!
```

Exercise

Write a method `union` for forming the union of two sets. You should implement the following abstract class.

```
abstract class IntSet:
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
  def union(other: IntSet): IntSet
end IntSet
```

Dynamic Binding

Object-oriented languages (including Scala) implement *dynamic method dispatch*.

This means that the code invoked by a method call depends on the runtime type of the object that contains the method.

Example

```
Empty.contains(1)
```

Dynamic Binding

Object-oriented languages (including Scala) implement *dynamic method dispatch*.

This means that the code invoked by a method call depends on the runtime type of the object that contains the method.

Example

```
Empty.contains(1)
```

→ `[1/x] [Empty/this] false`

Dynamic Binding

Object-oriented languages (including Scala) implement *dynamic method dispatch*.

This means that the code invoked by a method call depends on the runtime type of the object that contains the method.

Example

```
Empty.contains(1)
```

$\rightarrow$ `[1/x] [Empty/this] false`

$=$ `false`

Dynamic Binding (2)

Another evaluation using `NonEmpty`:

```
(NonEmpty(7, Empty, Empty)).contains(7)
```

Dynamic Binding (2)

Another evaluation using `NonEmpty`:

```
(NonEmpty(7, Empty, Empty)).contains(7)
```

$\rightarrow$ $[7/\texttt{elem}]$ $[7/\texttt{x}]$ $[\texttt{new NonEmpty(7, Empty, Empty)}/\texttt{this}]$

```
  if x < elem then this.left.contains(x)
    else if x > elem then this.right.contains(x) else true
```

Dynamic Binding (2)

Another evaluation using `NonEmpty`:

```
(NonEmpty(7, Empty, Empty)).contains(7)
```

$\rightarrow$ $[7/\texttt{elem}]$ $[7/\texttt{x}]$ $[\texttt{new NonEmpty(7, Empty, Empty)}/\texttt{this}]$

```
    if x < elem then this.left.contains(x)
      else if x > elem then this.right.contains(x) else true
```

$=$

```
if 7 < 7 then NonEmpty(7, Empty, Empty).left.contains(7)
    else if 7 > 7 then NonEmpty(7, Empty, Empty).right
        .contains(7) else true
```

Dynamic Binding (2)

Another evaluation using `NonEmpty`:

```
(NonEmpty(7, Empty, Empty)).contains(7)
```

$\rightarrow$ [7/elem] [7/x] [new NonEmpty(7, Empty, Empty)/this]

```
    if x < elem then this.left.contains(x)
      else if x > elem then this.right.contains(x) else true
```

$=$

```
if 7 < 7 then NonEmpty(7, Empty, Empty).left.contains(7)
    else if 7 > 7 then NonEmpty(7, Empty, Empty).right
        .contains(7) else true
```

$\rightarrow$ `true`

Something to Ponder

Dynamic dispatch of methods is analogous to calls to higher-order functions.

Question:

Can we implement one concept in terms of the other?

- Objects in terms of higher-order functions?
- Higher-order functions in terms of objects?