Case Classes vs Simple Classes

Effective Programming in Scala

Two Flavors of Type Definition

We have seen how to use classes and traits to achieve encapsulation.

We have previously been using case classes and sealed traits to model business domains.

This lesson shows the similarities and differences between both flavours of type definitions.

Case Classes vs Simple Classes

As we mentioned in a previous lecture, the main difference between case classes and classes is that the former achieve aggregation, whereas the latter achieve encapsulation.

An important consequence is that constructor parameters of case classes are promoted to public members, whereas constructor parameters of simple classes are, by default, private.

Although the purpose of aggregation and encapsulation are quite different, it makes sense to model both with the same concept, namely classes.

Actually, a case class is *just* a special case of a simple class.

Case Classes vs Simple Classes

When you define a case class, the compiler defines a class and customizes some parts of it.

- constructor parameters are promoted to public members (a.k.a. **fields**),
- an **extractor** enables pattern matching,
- **equality** operator between instances compares the values of the case class fields.

Structural vs Reference Equality

```
case class CaseClass(value: Int)

CaseClass(42) == CaseClass(42) // true
```

However, with default equality:

```
class SimpleClass(value: Int)

SimpleClass(42) == SimpleClass(42) // false
```

(We will further discuss this in the coming weeks)

Similarities Between Case Classes and Simple Classes

Despite these differences, both case classes and simple classes:

- define a new type along with a constructor,
- can have public and private members,
- can extend traits and override members,
- create abstraction levels.

Sealed Traits vs Simple Traits

The only difference between sealed traits and simple traits is that the former have a fixed number of concrete subclasses, whereas the latter have an unlimited number of subclasses.

More precisely, the subclasses of a sealed trait have to be defined in the same file as the sealed trait.

A practical consequence is that exhaustivity checking in pattern matching is only possible with sealed traits.

Similarities Between Sealed Traits and Simple Traits

Despite these differences, both sealed traits and simple traits:

- define a new type with no constructor,
- can have concrete and abstract members,
- can have public, protected, and private members,
- create abstraction levels.

Types: a Change of Perspective

Case classes and sealed traits make you think of types as **sets of possible values**. For instance, the type `Boolean` has two possible values, the type `Option[Boolean]` has three possible values, etc.

Alternatively, "simple" traits and classes make you think of types as **interfaces** that provide a specific set of operations. For instance, the type `Boolean` has logic operations such as `||` and `&&`.

Summary

Types can be seen as sets of operations (with an unbounded number of possible values), or sets of possible values (with an unbounded number of operations),

Classes conveniently group operations together.

If the set of possible values of a type is bounded, then it is probably a good idea to model this type with a case class.

If the set of possible operations on the values of some type is bounded, then it is probably a good idea to model this type with a class.