EPFL

Sealed Traits

Effective Programming in Scala

Accepting Alternative Values with Sealed Traits

Case classes **aggregate** several values into a single concept (e.g., a rectangle aggregates a width and a height).

Conversely, we can define types that **accept** only one of several alternative values.

For instance, we can define a type Shape that is either a Rectangle or a Circle as follows:

```
sealed trait Shape
case class Rectangle(width: Int, height: Int) extends Shape
case class Circle(radius: Int) extends Shape
```

Constructing Shapes

```scala
sealed trait Shape
case class Rectangle(width: Int, height: Int) extends Shape
case class Circle(radius: Int) extends Shape
```

Unlike case class definitions, sealed trait definitions don't introduce constructors. We say that traits are **abstract** types.

The only way to construct a value of type `Shape` is to use the constructor of a concrete class that extends `Shape`:

```scala
val someShape: Shape = Circle(5)
```

Digression: Subtyping

```
sealed trait Shape
case class Rectangle(width: Int, height: Int) extends Shape
case class Circle(radius: Int) extends Shape
```

It is worth noting that a value of type `Rectangle` or `Circle`, is also of type Shape.

```
val someRectangle: Rectangle = Rectangle(1, 2)
val someShape: Shape         = someRectangle
```

Digression: Subtyping

```scala
sealed trait Shape
case class Rectangle(width: Int, height: Int) extends Shape
case class Circle(radius: Int) extends Shape
```

It is worth noting that a value of type `Rectangle` or `Circle`, is also of type `Shape`.

```scala
val someRectangle: Rectangle = Rectangle(1, 2)
val someShape: Shape         = someRectangle
```

We say that types `Rectangle` and `Circle` are **subtypes** of `Shape`.

We will learn more about subtyping in the following weeks.

Manipulating Shapes

```
sealed trait Shape
case class Rectangle(width: Int, height: Int) extends Shape
case class Circle(radius: Int) extends Shape
```

The type Shape has no members.

The only (useful) thing we can do with a value of type Shape is to recover its concrete type using a **match** expression.

```
val someShapeArea =
  someShape match
    case Rectangle(width, height) => width * height
    case Circle(radius)           => radius * radius * 3.14
```

Match Expressions

```
val someShapeArea =
  someShape match
    case Rectangle(width, height) => width * height
    case Circle(radius)           => radius * radius * 3.14
```

In this example, we define the result of `someShapeArea` by listing all the possible cases of `Shape`.

In case `someShape` is a `Rectangle`, then the result of the `match` expression is `width * height`, where these values are **extracted** from the underlying rectangle.

Match Expressions (Scala 2 Compatibility)

In Scala 2, the “cases” in a match expression had to be surrounded by braces (indentation was not significant):

```
val someShapeArea =
  someShape match {
    case Rectangle(width, height) => width * height
    case Circle(radius)           => radius * radius * 3.14
  }
```

Scala 3 still supports this syntax.

Exhaustivity Checking

What happens if you compile a match expression that doesn't cover all the cases?

```scala
val someShapeArea =
  someShape match
    case Rectangle(width, height) => width * height
```

Exhaustivity Checking

What happens if you compile a match expression that doesn't cover all the cases?

```scala
val someShapeArea =
  someShape match
    case Rectangle(width, height) => width * height
```

The compiler warns you and provides examples of input that would fail to be matched by your patterns:

```
[warn] Match may not be exhaustive.
[warn] It would fail on the following input: Circle(_)
```

Alternative Patterns

So far we have shown how to recover the concrete type of a value and to extract its fields. There are other things we can do when we match on expressions. Here are two alternative patterns that you might find useful.

```
someShape match
  case circle: Circle => s"This is a circle with radius ${circle.radius}"
  case _              => "This is not a circle"
```

- The **typed pattern** matches only instances of type `Circle` and binds the name `circle` to the matched value,
- The **wildcard pattern** matches everything.

Exercise

Model the actions that a user can perform in a messaging system:

- *subscribe to a channel,*
- *unsubscribe from a channel,*
- *or post a message to a channel.*

A channel has a name.

Exercise

Model the actions that a user can perform in a messaging system:

- *subscribe to a channel,*
- *unsubscribe from a channel,*
- *or post a message to a channel.*

A channel has a name.

```scala
sealed trait Action
case class Subscribe(channel: Channel) extends Action
case class Unsubscribe(channel: Channel) extends Action
case class PostMessage(channel: Channel, message: String) extends Action

case class Channel(name: String)
```

Exercise

Define a value representing the action of subscribing to a channel named "effective-scala"

Exercise

Define a value representing the action of subscribing to a channel named "effective-scala"

```
val subscribeEffectiveScala =
  Subscribe(Channel("effective-scala"))
```

Summary

A *case class* aggregates several concepts together, whereas a *sealed trait* represents one of several alternatives.

These two building blocks can be used to *model* business domains.

Match expressions can be used to define alternative *branches* of a program according to the concrete class of a sealed trait, and to *extract* data from this class at the same time.