Class Hierarchies

CS-214 Software Construction

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())
```

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` include `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` instance can be (or needs 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)
```

$\rightarrow$ `[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/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
```

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
```

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?

How Classes are Organized

CS-214 Software Construction

Packages

Classes and objects are organized in packages.

To place a class or object inside a package, use a package clause at the top of your source file.

```
package progfun.examples

object Hello
  ...
```

This would place `Hello` in the package `progfun.examples`.

You can then refer it by its *fully qualified name*, `progfun.examples.Hello`. For instance, to run the `Hello` program:

```
> scala progfun.examples.Hello
```

Imports

Say we have a class `Rational` in package `week3`.

You can use the class using its fully qualified name:

```
val r = week3.Rational(1, 2)
```

Alternatively, you can use an import:

```
import week3.Rational
val r = Rational(1, 2)
```

Forms of Imports

Imports come in several forms:

```
import week3.Rational           // imports just Rational
import week3.{Rational, Hello}  // imports both Rational and Hello
import week3.*                  // imports everything in package week3
```

The first two forms are called *named imports*.

The last form is called a *wildcard import*.

You can import from either a package or an object.

Automatic Imports

Some entities are automatically imported in any Scala program.

These are:

- All members of package `scala`
- All members of package `java.lang`
- All members of the singleton object `scala.Predef`.

Here are the fully qualified names of some types and functions which you have seen so far:

`Int`	`scala.Int`
`Boolean`	`scala.Boolean`
`Object`	`java.lang.Object`
`require`	`scala.Predef.require`
`assert`	`scala.Predef.assert`

Scaladoc

You can explore the standard Scala library using the scaladoc web pages.

You can start at

www.scala-lang.org/api/current

Traits

In Java, as well as in Scala, a class can only have one superclass.

But what if a class has several natural supertypes to which it conforms or from which it wants to inherit code?

Here, you could use `traits`.

A trait is declared like an abstract class, just with `trait` instead of `abstract class`.

```
trait Planar:
  def height: Int
  def width: Int
  def surface = height * width
```

Traits (2)

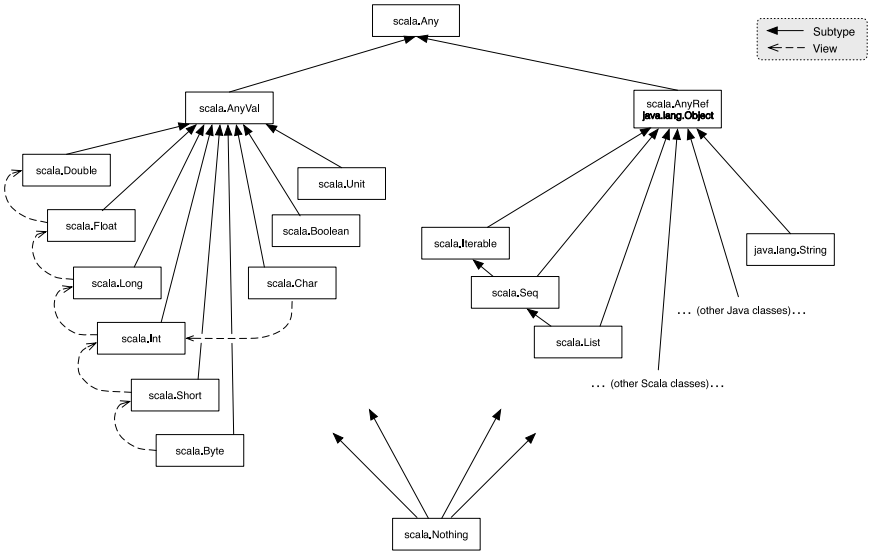
Classes, objects and traits can inherit from at most one class but arbitrary many traits.

Example:

```
class Square extends Shape, Planar, Movable ...
```

Traits resemble interfaces in Java, but are more powerful because they can have parameters and can contain fields and concrete methods.

Scala's Class Hierarchy



Top Types

At the top of the type hierarchy we find:

`Any`	the base type of all types Methods: ‘==‘, ‘!=‘, ‘equals‘, ‘hashCode, ‘toString‘
`AnyRef`	The base type of all reference types; Alias of ‘java.lang.Object‘
`AnyVal`	The base type of all primitive types.

The Nothing Type

`Nothing` is at the bottom of Scala's type hierarchy. It is a subtype of every other type.

There is no value of type `Nothing`.

Why is that useful?

- To signal abnormal termination
- As an element type of empty collections (see next session)

Exceptions

Scala's exception handling is similar to Java's.

The expression

```
throw Exc
```

aborts evaluation with the exception `Exc`.

The type of this expression is `Nothing`.

Objects Everywhere (Optional Material)

CS-214 Software Construction

Pure Object Orientation

A pure object-oriented language is one in which every value is an object.

If the language is based on classes, this means that the type of each value is a class.

Is Scala a pure object-oriented language?

At first glance, there seem to be some exceptions: primitive types, functions.

But, let's look closer:

Standard Classes

Conceptually, types such as `Int` or `Boolean` do not receive special treatment in Scala. They are like the other classes, defined in the package `scala`.

For reasons of efficiency, the Scala compiler represents the values of type `scala.Int` by 32-bit integers, and the values of type `scala.Boolean` by Java's Booleans, etc.

Pure Booleans

The `Boolean` type maps to the JVM's primitive type `boolean`.

But one *could* define it as a class from first principles:

```
package idealized.scala
abstract class Boolean extends AnyVal:
  def ifThenElse[T](t: => T, e: => T): T

  def && (x: => Boolean): Boolean = ifThenElse(x, false)
  def || (x: => Boolean): Boolean = ifThenElse(true, x)
  def unary_!: Boolean            = ifThenElse(false, true)

  def == (x: Boolean): Boolean    = ifThenElse(x, x.unary_!)
  def != (x: Boolean): Boolean    = ifThenElse(x.unary_!, x)
  ...
end Boolean
```

Boolean Constants

Here are constants `true` and `false` that go with `Boolean` in `idealized.scala`:

```
package idealized.scala

object true extends Boolean:
  def ifThenElse[T](t: => T, e: => T) = t

object false extends Boolean:
  def ifThenElse[T](t: => T, e: => T) = e
```

Exercise

Provide an implementation of an implication operator ==> for class `idealized.scala.Boolean`.

Exercise

Provide an implementation of an implication operator `==>` for class `idealized.scala.Boolean`.

```
extension (x: Boolean)
  def ==> (y: Boolean) = x.ifThenElse(y, true)
```

That is, if `x` is true, `y` has to be true also, whereas if `x` is false, `y` can be arbitrary.

The class Int

Here is a partial specification of the class `scala.Int`.

```
class Int:
  def + (that: Double): Double
  def + (that: Float): Float
  def + (that: Long): Long
  def + (that: Int): Int          // same for -, *, /, %

  def << (cnt: Int): Int          // same for >>, >>>  */

  def & (that: Long): Long
  def & (that: Int): Int          // same for |, ^ */
```

The class Int (2)

```
  def == (that: Double): Boolean
  def == (that: Float): Boolean
  def == (that: Long): Boolean   // same for !=, <, >, <=, >=
  ...
end Int
```

Can it be represented as a class from first principles (i.e. not using primitive `int`s?

Exercise

Provide an implementation of the abstract class `Nat` that represents non-negative integers.

```
abstract class Nat:
  def isZero: Boolean
  def predecessor: Nat
  def successor: Nat
  def + (that: Nat): Nat
  def - (that: Nat): Nat
end Nat
```

Exercise (2)

Do not use standard numerical classes in this implementation.

Rather, implement a sub-object and a sub-class:

```
object Zero extends Nat:
  ...
class Succ(n: Nat) extends Nat:
  ...
```

One for the number zero, the other for strictly positive numbers.

(this one is a bit more involved than previous quizzes).

Functions as Objects

CS-214 Software Construction

Functions as Objects

We have seen that Scala's numeric types and the `Boolean` type can be implemented like normal classes.

But what about functions?

Functions as Objects

We have seen that Scala's numeric types and the `Boolean` type can be implemented like normal classes.

But what about functions?

In fact function values *are* treated as objects in Scala.

The function type `A => B` is just an abbreviation for the class `scala.Function1[A, B]`, which is defined as follows.

```
package scala
trait Function1[A, B]:
  def apply(x: A): B
```

So functions are objects with `apply` methods.

There are also traits `Function2`, `Function3`, ... for functions which take more parameters.

Expansion of Function Values

An anonymous function such as

```
(x: Int) => x * x
```

is expanded to:

Expansion of Function Values

An anonymous function such as

```
(x: Int) => x * x
```

is expanded to:

```
new Function1[Int, Int]:
  def apply(x: Int) = x * x
```

Expansion of Function Values

An anonymous function such as

```
(x: Int) => x * x
```

is expanded to:

```
new Function1[Int, Int]:
  def apply(x: Int) = x * x
```

This *anonymous class* can itself be thought of as a block that defines and instantiates a local class:

```
{ class $anonfun() extends Function1[Int, Int]:
    def apply(x: Int) = x * x
  $anonfun()
}
```

Expansion of Function Calls

A function call, such as `f(a, b)`, where `f` is a value of some class type, is expanded to

```
f.apply(a, b)
```

So the OO-translation of

```
val f = (x: Int) => x * x
f(7)
```

would be

```
val f = new Function1[Int, Int]:
  def apply(x: Int) = x * x

f.apply(7)
```

Functions and Methods

Note that a method such as

```
def f(x: Int): Boolean = ...
```

is not itself a function value.

But if `f` is used in a place where a Function type is expected, it is converted automatically to the function value

```
(x: Int) => f(x)
```

or, expanded:

```
new Function1[Int, Boolean]:
  def apply(x: Int) = f(x)
```

Exercise

In package `week3`, define an

```
object IntSet:
  ...
```

with 3 functions in it so that users can create `IntSet`s of lengths 0-2 using syntax

```
IntSet()       // the empty set
IntSet(1)      // the set with single element 1
IntSet(2, 3)   // the set with elements 2 and 3.
```

Decomposition

CS-214 Software Construction

Decomposition

Suppose you want to write a small interpreter for arithmetic expressions.

To keep it simple, let's restrict ourselves to numbers and additions.

Expressions can be represented as a class hierarchy, with a base trait `Expr` and two subclasses, `Number` and `Sum`.

To treat an expression, it's necessary to know the expression's shape and its components.

This brings us to the following implementation.

Expressions

```
trait Expr:
  def isNumber: Boolean
  def isSum: Boolean
  def numValue: Int
  def leftOp: Expr
  def rightOp: Expr

class Number(n: Int) extends Expr:
  def isNumber = true
  def isSum = false
  def numValue = n
  def leftOp = throw Error("Number.leftOp")
  def rightOp = throw Error("Number.rightOp")
```

Expressions (2)

```
class Sum(e1: Expr, e2: Expr) extends Expr:
  def isNumber = false
  def isSum = true
  def numValue = throw Error("Sum.numValue")
  def leftOp = e1
  def rightOp = e2
```

Evaluation of Expressions

You can now write an evaluation function as follows.

```
def eval(e: Expr): Int =
  if e.isNumber then e.numValue
  else if e.isSum then eval(e.leftOp) + eval(e.rightOp)
  else throw Error("Unknown expression " + e)
```

Problem: Writing all these classification and accessor functions quickly becomes tedious!

Problem: There's no static guarantee you use the right accessor functions. You might hit an `Error` case if you are not careful.

Adding New Forms of Expressions

So, what happens if you want to add new expression forms, say

```
class Prod(e1: Expr, e2: Expr) extends Expr   // e1 * e2
class Var(x: String) extends Expr             // Variable ‘x’
```

You need to add methods for classification and access to all classes defined above.

Question

To integrate `Prod` and `Var` into the hierarchy, how many new method definitions do you need?

(including method definitions in `Prod` and `Var` themselves, but not counting methods that were already given on the slides)

Possible Answers

0	9
0	10
0	19
0	25
0	35
0	40

Question

To integrate `Prod` and `Var` into the hierarchy, how many new method definitions do you need?

(including method definitions in `Prod` and `Var` themselves, but not counting methods that were already given on the slides)

Possible Answers

0	9
0	10
0	19
X	25
X	35
0	40

(depending on whether `leftOp` and `rightOp` is shared between `Sum` and `Prod`).

Non-Solution: Type Tests and Type Casts

A "hacky" solution could use type tests and type casts.

Scala let's you do these using methods defined in class `Any`:

```
def isInstanceOf[T]: Boolean  // checks whether this object's type conforms to ‘T‘
def asInstanceOf[T]: T        // treats this object as an instance of type ‘T‘
                              // throws ‘ClassCastException‘ if it isn't.
```

These correspond to Java's type tests and casts

Scala	Java
`x.isInstanceOf[T]`	`x instanceof T`
`x.asInstanceOf[T]`	`(T) x`

But their use in Scala is discouraged, because there are better alternatives.

Eval with Type Tests and Type Casts

Here's a formulation of the `eval` method using type tests and casts:

```
def eval(e: Expr): Int =
  if e.isInstanceOf[Number] then
    e.asInstanceOf[Number].numValue
  else if e.isInstanceOf[Sum] then
    eval(e.asInstanceOf[Sum].leftOp)
    + eval(e.asInstanceOf[Sum].rightOp)
  else throw Error("Unknown expression " + e)
```

This is ugly and potentially unsafe.

Solution 1: Object-Oriented Decomposition

For example, suppose that all you want to do is *evaluate* expressions.

You could then define:

```
trait Expr:
  def eval: Int

class Number(n: Int) extends Expr:
  def eval: Int = n

class Sum(e1: Expr, e2: Expr) extends Expr:
  def eval: Int = e1.eval + e2.eval
```

But what happens if you'd like to display expressions now?

You have to define new methods in all the subclasses.

Assessment of OO Decomposition

- OO decomposition mixes *data* with *operations* on the data.
- This can be the right thing if there's a need for encapsulation and data abstraction.
- On the other hand, it increases complexity(*) and adds new dependencies to classes.
- It makes it easy to add new kinds of data but hard to add new kinds of operations.

(*) In the literal sense of the word:
complex = plaited, woven together

Thus, complexity arises from mixing several things together.

Limitations of OO Decomposition

OO decomposition only works well if operations are on a *single* object.

What if you want to simplify expressions, say using the rule:

```
a * b + a * c   ->   a * (b + c)
```

Problem: This is a non-local simplification. It cannot be encapsulated in the method of a single object.

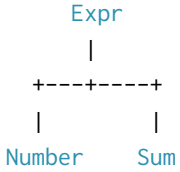
You are back to square one; you need test and access methods for all the different subclasses.

Pattern Matching

CS-214 Software Construction

Reminder: Decomposition

The task we are trying to solve is find a general and convenient way to access heterogeneous data in a class hierarchy.



Attempts seen previously:

- *Classification and access methods*: quadratic explosion
- *Type tests and casts*: unsafe, low-level
- *Object-oriented decomposition*: causes coupling between data and operations, need to touch all classes to add a new method.

Solution 2: Functional Decomposition with Pattern Matching

Observation: the sole purpose of test and accessor functions is to *reverse* the construction process:

- Which subclass was used?
- What were the arguments of the constructor?

This situation is so common that many functional languages, Scala included, automate it.

Case Classes

A *case class* definition is similar to a normal class definition, except that it is preceded by the modifier `case`. For example:

```
trait Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
```

Like before, this defines a trait `Expr`, and two concrete subclasses `Number` and `Sum`.

However, these classes are now empty. So how can we access the members?

Pattern Matching

Pattern matching is a generalization of `switch` from C/Java to class hierarchies.

It's expressed in Scala using the keyword `match`.

Example

```
def eval(e: Expr): Int = e match
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
```

Match Syntax

Rules:

- `match` is preceded by a selector expression and is followed by a sequence of *cases*, `pat => expr`.
- Each case associates an *expression* `expr` with a *pattern* `pat`.
- A `MatchError` exception is thrown if no pattern matches the value of the selector.

Forms of Patterns

Patterns are constructed from:

- *constructors*, e.g. `Number`, `Sum`,
- *variables*, e.g. `n`, `e1`, `e2`,
- *wildcard patterns* `_`,
- *constants*, e.g. `1`, `true`.
- *type tests*, e.g. `n: Number`

Variables always begin with a lowercase letter.

The same variable name can only appear once in a pattern. So, `Sum(x, x)` is not a legal pattern.

Names of constants begin with a capital letter, with the exception of the reserved words `null`, `true`, `false`.

Evaluating Match Expressions

An expression of the form

$$\texttt{e match } \{ \texttt{ case } p_1 \texttt{ => } e_1 \texttt{ ... case } p_n \texttt{ => } e_n \texttt{ } \}$$

matches the value of the selector e with the patterns $p_1, ..., p_n$ in the order in which they are written.

The whole match expression is rewritten to the right-hand side of the first case where the pattern matches the selector *e*.

References to pattern variables are replaced by the corresponding parts in the selector.

What Do Patterns Match?

- A constructor pattern $C(p_1, ..., p_n)$ matches all the values of type C (or a subtype) that have been constructed with arguments matching the patterns $p_1, ..., p_n$.
- A variable pattern x matches any value, and *binds* the name of the variable to this value.
- A constant pattern c matches values that are equal to c (in the sense of ==)

Example

Example

```
eval(Sum(Number(1), Number(2)))
```

→

```
Sum(Number(1), Number(2)) match
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
```

→

```
eval(Number(1)) + eval(Number(2))
```

Example (2)

→

```
Number(1) match
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
+ eval(Number(2))
```

→

```
1 + eval(Number(2))
```

↠

```
3
```

Pattern Matching and Methods

Of course, it's also possible to define the evaluation function as a method of the base trait.

Example

```
trait Expr:
  def eval: Int = this match
    case Number(n) => n
    case Sum(e1, e2) => e1.eval + e2.eval
```

Enums

CS-214 Software Construction

Pure Data

In the previous sessions, you have learned how to model data with class hierarchies.

Classes are essentially bundles of functions operating on some common values represented as fields.

They are a very useful abstraction, since they allow encapsulation of data.

But sometimes we just need to compose and decompose *pure data* without any associated functions.

Case classes and pattern matching work well for this task.

A Case Class Hierarchy

Here's our case class hierarchy for expressions again:

```
trait Expr
object Expr:
  case class Var(s: String) extends Expr
  case class Number(n: Int) extends Expr
  case class Sum(e1: Expr, e2: Expr) extends Expr
  case class Prod(e1: Expr, e2: Expr) extends Expr
```

This time we have put all case classes in the `Expr` companion object, in order not to pollute the global namespace.

So it's `Expr.Number(1)` instead of `Number(1)`, for example.

One can still "pull out" all the cases using an import.

```
import Expr.*
```

A Case Class Hierarchy

Here's our case class hierarchy for expressions again:

```
trait Expr
object Expr:
  case class Var(s: String) extends Expr
  case class Number(n: Int) extends Expr
  case class Sum(e1: Expr, e2: Expr) extends Expr
  case class Prod(e1: Expr, e2: Expr) extends Expr
```

Pure data definitions like these are called *algebraic data types*, or ADTs for short.

They are very common in functional programming.

To make them even more convenient, Scala offers some special syntax.

Enums for ADTs

An *enum* enumerates all the cases of an ADT *and nothing else*.

Example

```
enum Expr:
  case Var(s: String)
  case Number(n: Int)
  case Sum(e1: Expr, e2: Expr)
  case Prod(e1: Expr, e2: Expr)
```

This enum is equivalent to the case class hierarchy on the previous slide, but is shorter, since it avoids the repetitive `class ... extends Expr` notation.

Pattern Matching on ADTs

Match expressions can be used on enums as usual.

For instance, to print expressions with proper parameterization:

```
def show(e: Expr): String = e match
  case Expr.Var(x) => x
  case Expr.Number(n) => n.toString
  case Expr.Sum(a, b) => s"${show(a)} + ${show(b)}"
  case Expr.Prod(a, b) => s"${showP(a)} * ${showP(b)}"

def showP(e: Expr): String = e match
  case e: Sum => s"(${show(e)})"
  case _ => show(e)
```

Simple Enums

Cases of an `enum` can also be simple values, without any parameters.

Example

Define a `Color` type with values `Red`, `Green`, and `Blue`:

```
enum Color:
  case Red
  case Green
  case Blue
```

We can also combine several simple cases in one list:

```
enum Color:
  case Red, Green, Blue
```

Pattern Matching on Simple Enums

For pattern matching, simple cases count as constants:

```
enum DayOfWeek:
  case Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday

import DayOfWeek.*

def isWeekend(day: DayOfWeek) = day match
  case Saturday | Sunday => true
  case _ => false
```

More Fun With Enums

Enumerations can take parameters and can define methods.

Example:

```
enum Direction(val dx: Int, val dy: Int):
  case Right extends Direction( 1,  0)
  case Up    extends Direction( 0,  1)
  case Left  extends Direction(-1,  0)
  case Down  extends Direction( 0, -1)

  def leftTurn = Direction.values((ordinal + 1) % 4)
end Direction

val r = Direction.Right
val u = x.leftTurn       // u = Up
val v = (u.dx, u.dy)     // v = (1, 0)
```

More Fun With Enums

Notes:

- Enumeration cases that pass parameters have to use an explicit `extends` clause
- The expression `e.ordinal` gives the ordinal value of the enum case `e`. Cases start with zero and are numbered consecutively.
- `values` is an immutable array in the companion object of an enum that contains all enum values.
- Only simple cases have `ordinal` numbers and show up in `values`, parameterized cases do not.

Enumerations Are Shorthands for Classes and Objects

The `Direction` enum is expanded by the Scala compiler to roughly the following structure:

```
abstract class Direction(val dx: Int, val dy: Int):
  def leftTurn = Direction.values((ordinal + 1) % 4)
object Direction:
  val Right = new Direction( 1,  0) {}
  val Up    = new Direction( 0,  1) {}
  val Left  = new Direction(-1,  0) {}
  val Down  = new Direction( 0, -1) {}
end Direction
```

There are also compiler-defined helper methods `ordinal` in the class and `values` and `valueOf` in the companion object.

Domain Modeling

ADTs and enums are particularly useful for domain modelling tasks where one needs to define a large number of data types without attaching operations.

Example: Modelling payment methods.

```
enum PaymentMethod:
  case CreditCard(kind: Card, holder: String, number: Long, expires: Date)
  case PayPal(email: String)
  case Cash

enum Card:
  case Visa, Mastercard, Amex
```

Summary

In this unit, we covered two uses of `enum` definitions:

- as a shorthand for hierarchies of case classes,
- as a way to define data types accepting alternative values,

The two cases can be combined: an enum can comprise parameterized and simple cases at the same time.

Enums are typically used for pure data, where all operations on such data are defined elsewhere.