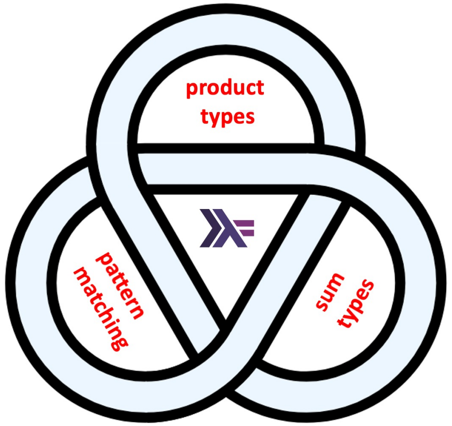
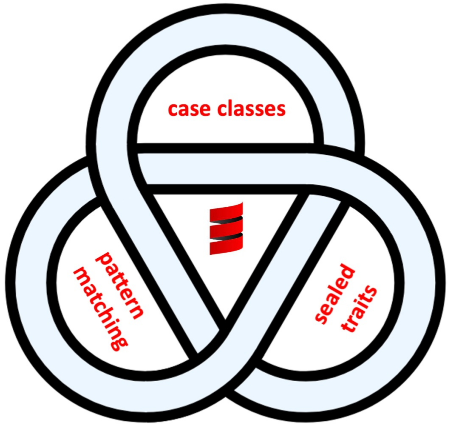


Algebraic Data Types for Data Oriented Programming



@BrianGoetz
Java Language Architect

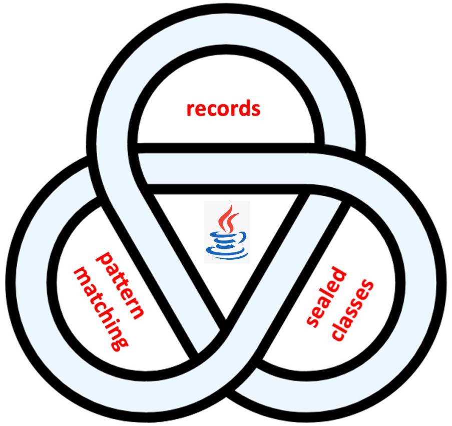
From **Haskell** and **Scala** to **Java**

Inspired by and based on **Brian Goetz**’s blog post

Data Oriented Programming in Java

InfoQ queue

https://www.infoq.com/articles/data-oriented-programming-java/



slides by  @philip_schwarz slideshare https://www.slideshare.net/pjschwarz

@philip_schwarz

This slide deck was inspired by **Brian Goetz**'s great InfoQ blog post **Data Oriented Programming in Java**.

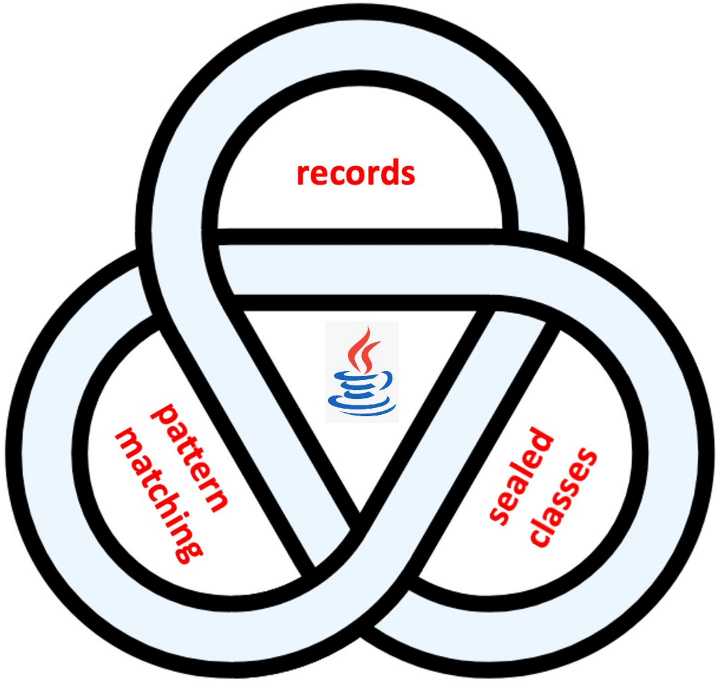
I really liked the post and found it very useful. It is great to see **Java** finally supporting **Data Oriented Programming**.

The first three slides of the deck consist of excerpts from the post.

InfoQueue

https://www.infoq.com/articles/data-oriented-programming-java/

@BrianGoetz



Data-oriented programming

Java's strong static typing and class-based modeling can still be tremendously useful for smaller programs, just in different ways. **Where OOP encourages us to use classes to model business entities and processes, smaller codebases with fewer internal boundaries will often get more mileage out of using classes to model *data*.** Our services consume requests that come from the outside world, such as via HTTP requests with untyped JSON/XML/YAML payloads. But only the most trivial of services would want to work directly with data in this form; we'd like to represent numbers as int or long rather than as strings of digits, dates as classes like LocalDateTime, and lists as collections rather than long comma-delimited strings. (And we want to validate that data at the boundary, before we act on it.)

> **Data-oriented programming encourages us to model *data as data*. Records, sealed classes, and pattern matching, work together to make that easier.**

Data-oriented programming encourages us to model data as (immutable) data, and keep the code that embodies the business logic of how we act on that data separately. As this trend towards smaller programs has progressed, **Java has acquired new tools to make it easier to model data as data** (**records**), **to directly model alternatives** (**sealed classes**), **and to flexibly destructure polymorphic data** (**pattern matching**) **patterns**.

Programming with data as data doesn't mean giving up static typing. One *could* do data-oriented programming with only untyped maps and lists (one often does in languages like Javascript), but static typing still has a lot to offer in terms of safety, readability, and maintainability, even when we are only modeling plain data. (Undisciplined data-oriented code is often called "stringly typed", because it uses strings to model things that shouldn't be modeled as strings, such as numbers, dates, and lists.)

@BrianGoetz

The combination of records, sealed types, and pattern matching makes it easy to follow these principles, yielding more concise, readable, and more reliable programs.

While programming with data as data may be a little unfamiliar given Java's OO underpinnings, these techniques are well worth adding to our toolbox.

Algebraic data types

This combination of records and sealed types is an example of what are called *algebraic data types* (ADTs). Records are a form of "**product types**", so-called because their state space is the **cartesian product** of that of their components. Sealed classes are a form of "**sum types**", so-called because the set of possible values is the sum (**union**) of the value sets of the alternatives. **This simple combination of mechanisms** -- **aggregation** and **choice** -- **is deceptively powerful, and shows up in many programming languages.**

It's not either/or

Many of the ideas outlined here may look, at first, to be somewhat "**un-Java-like**", because most of us have been taught to start by modeling entities and processes as **objects**. But in reality, our programs often work with relatively simple data, which often comes from the "outside world" where we can't count on it fitting cleanly into the Java type system. …

When we're modeling complex entities, or writing rich libraries such as java.util.stream, **OO** techniques have a lot to offer us. But when we're building simple services that process plain, **ad-hoc data**, the techniques of **data-oriented programming** may offer us a straighter path. Similarly, when exchanging complex results across an API boundary (such as our match result example), it is often simpler and clearer to define an ad-hoc data schema using **ADT**s, than to complect results and behavior in a stateful object (as the Java Matcher API does.)

The techniques of OOP and data-oriented programming are not at odds; they are different tools for different granularities and situations. We can freely mix and match them as we see fit.

@BrianGoetz

Data oriented programming in Java

Records, sealed classes, and pattern matching are designed to work together to support data-oriented programming.

Records allow us to simply model data using classes; sealed classes let us model *choices*; and pattern matching provides us with an easy and type-safe way of acting on polymorphic data.

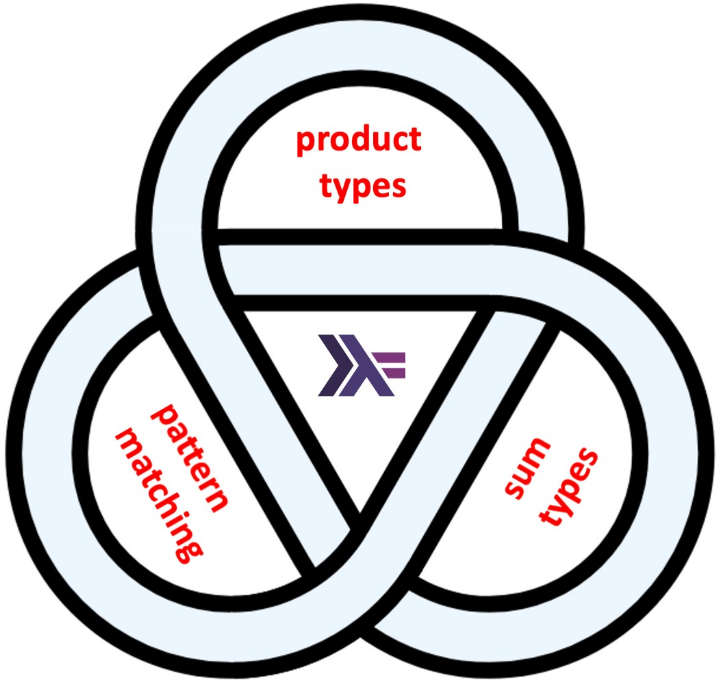
Support for **pattern matching** has come in several increments; the first added only type-test patterns and only supported them in **instanceof**; the next supported type-test patterns in switch as well; and most recently, *deconstruction patterns for records* were added in **Java 19**. The examples in this article will make use of all of these features.

While **records** are syntactically concise, their main strength is that they let us cleanly and simply model **aggregates**.

Just as with all data modeling, there are creative decisions to make, and some modelings are better than others.

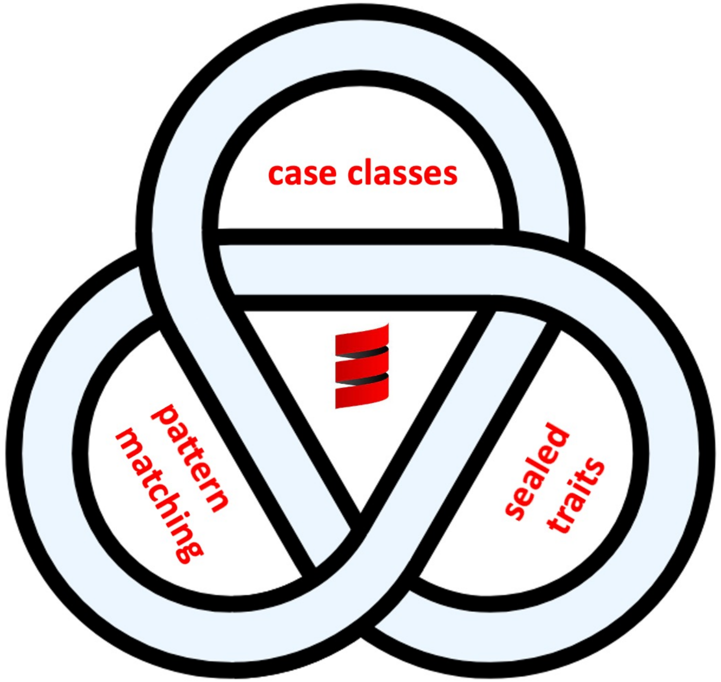
Using the combination of records and sealed classes also makes it easier to *make illegal states unrepresentable*, further improving safety and maintainability.

Among **Haskell**, **Scala** and **Java**, **Haskell** was the first to include features enabling **Data Oriented programming**. When **Scala** was born, it also supported the paradigm. In Java, support for the paradigm is being retrofitted.


product types

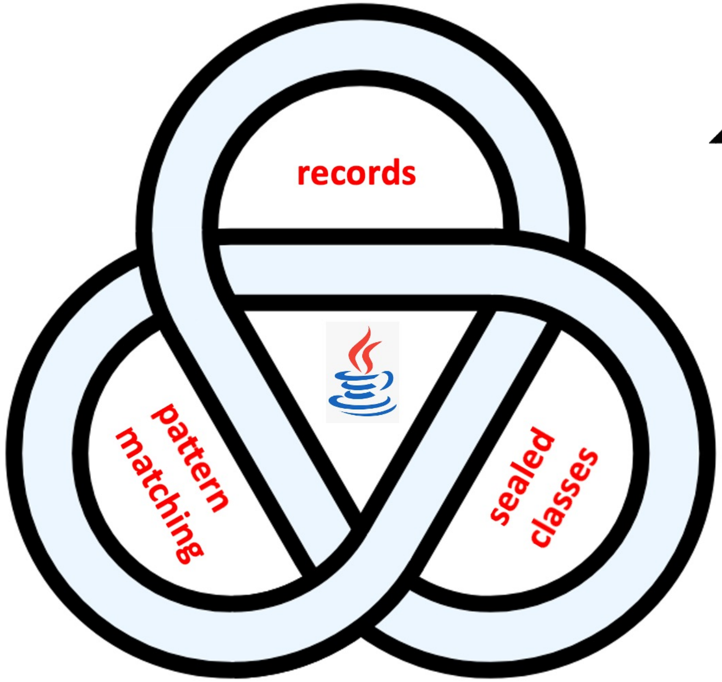
pattern matching

sum types


case classes

pattern matching

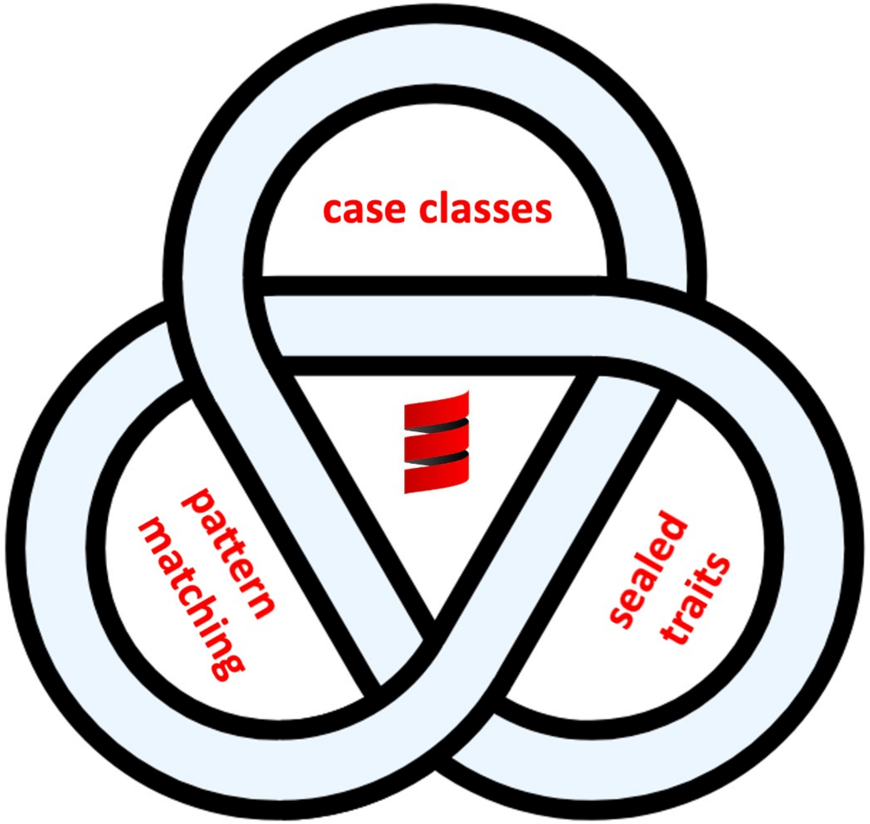
sealed traits

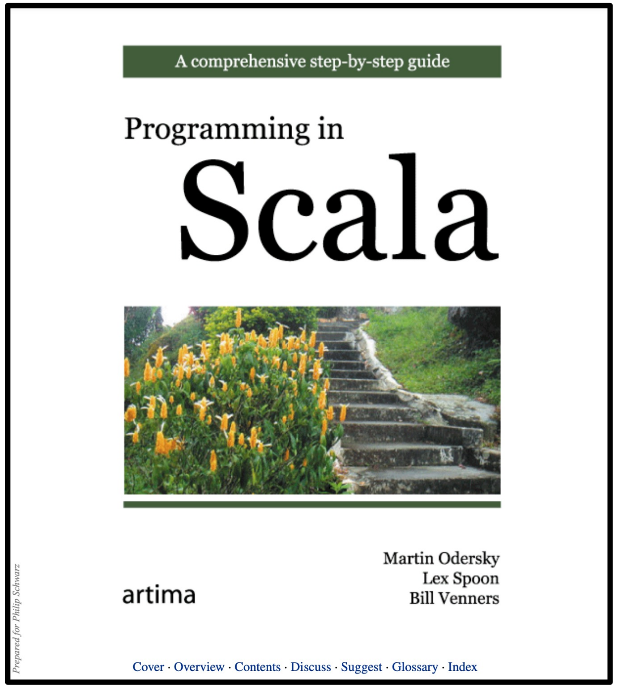

records

pattern matching

sealed classes

In the next three slides we look at how, since its inception, **Scala** included features enabling **Data Oriented programming**: **case classes**, **sealed abstract classes** (or **sealed traits**), and **pattern matching**.





A comprehensive step-by-step guide

Programming in
Scala

Martin Odersky
Lex Spoon
Bill Venners

artima

Prepared for Philip Schwarz

Cover · Overview · Contents · Discuss · Suggest · Glossary · Index

2007

Case classes

The other noteworthy thing about the declarations of Listing 15.1 is that each subclass has a case modifier. Classes with such a modifier are called case classes. Using the modifier makes the Scala compiler add some syntactic conveniences to your class.

Chapter 15 Case Classes and Pattern Matching

This chapter introduces case classes and pattern matching, twin constructs that support you when writing regular, non-encapsulated data structures. These two constructs are particularly helpful for tree-like recursive data. If you have programmed in a functional language before, then you will probably recognize pattern matching. Case classes will be new to you, though. Case classes are Scala's way to allow pattern matching on objects without requiring a large amount of boilerplate. In the common case, all you need to do is add a single case keyword to each class that you want to be pattern matchable. This chapter starts with a simple example of case classes and pattern matching. It then goes through all of the kinds of patterns that are supported, talks about the role of sealed classes, discusses the Option type, and shows some non-obvious places in the language where pattern matching is used. Finally, a larger, more realistic example of pattern matching is shown.

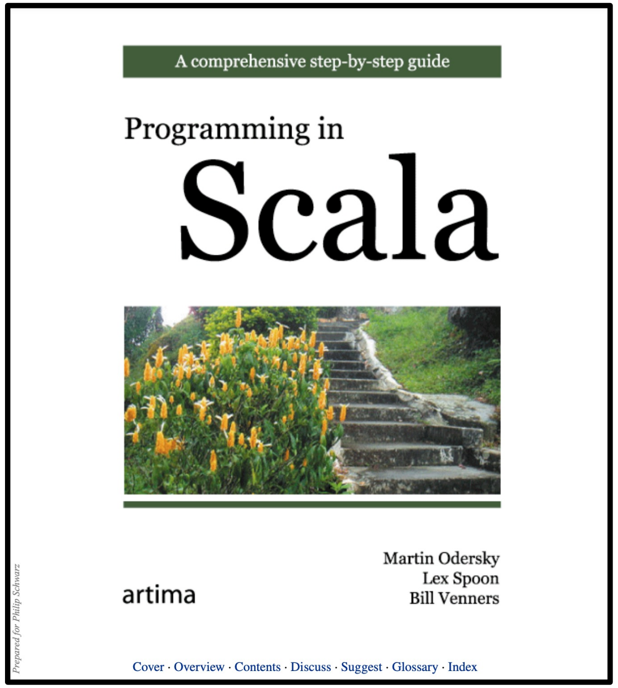
15.1 A simple example

Before delving into all the rules and nuances of pattern matching, it is worth looking at a simple example to get the general idea. Let's say you need to write a library that manipulates arithmetic expressions, perhaps as part of a domain-specific language you are designing. A first step to tackle this problem is the definition of the input data. To keep things simple, we'll concentrate on arithmetic expressions consisting of variables, numbers, and unary and binary operations. This is expressed by the hierarchy of Scala classes shown in Listing 15.1.

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr) extends Expr
```

Listing 15.1 · Defining case classes.

The hierarchy includes an abstract base class Expr with four subclasses, one for each kind of expression being considered. The bodies of all five classes are empty. As mentioned previously, in Scala you can leave out the braces around an empty class body if you wish, so class C is the same as class C {}.


A comprehensive step-by-step guide

Programming in Scala

Martin Odersky
Lex Spoon
Bill Venners

artima

Cover · Overview · Contents · Discuss · Suggest · Glossary · Index

Prepared for Philip Schwarz

2007

15.5 Sealed classes

Whenever you write a pattern match, you need to make sure you have covered all of the possible cases. Sometimes you can do this by adding a default case at the end of the match, but that only applies if there is a sensible default behavior. What do you do if there is no default? How can you ever feel safe that you covered all the cases?

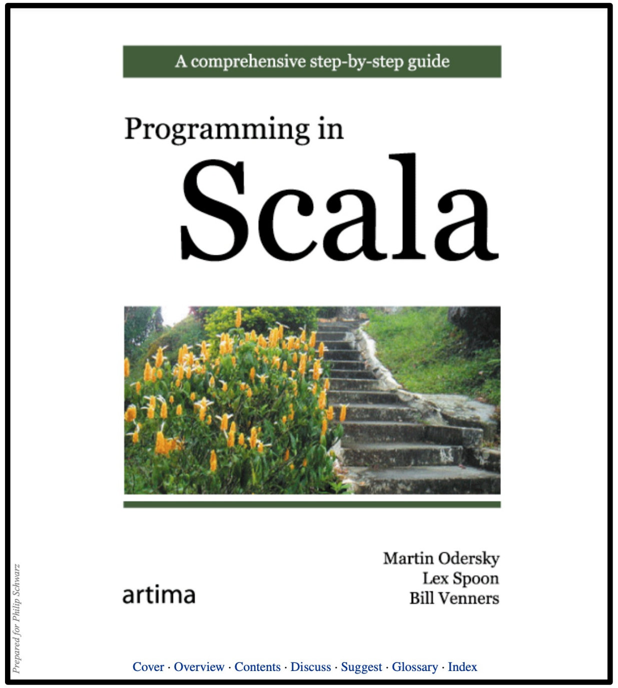
In fact, you can enlist the help of the Scala compiler in detecting missing combinations of patterns in a match expression. To be able to do this, the compiler needs to be able to tell which are the possible cases. In general, this is impossible in Scala, because new case classes can be defined at any time and in arbitrary compilation units. For instance, nothing would prevent you from adding a fifth case class to the `Expr` class hierarchy in a different compilation unit from the one where the other four cases are defined.

The alternative is to make the superclass of your case classes sealed. A sealed class cannot have any new subclasses added except the ones in the same file. This is very useful for pattern matching, because it means you only need to worry about the subclasses you already know about. What's more, you get better compiler support as well. If you match against case classes that inherit from a sealed class, the compiler will flag missing combinations of patterns with a warning message.

Therefore, if you write a hierarchy of classes intended to be pattern matched, you should consider sealing them. Simply put the sealed keyword in front of the class at the top of the hierarchy. Programmers using your class hierarchy will then feel confident in pattern matching against it. The sealed keyword, therefore, is often a license to pattern match. Listing 15.16 shows an example in which `Expr` is turned into a sealed class.

```
sealed abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr) extends Expr
```

Listing 15.16 · A sealed hierarchy of case classes.



A comprehensive step-by-step guide

Programming in

Scala

Martin Odersky
Lex Spoon
Bill Venners

artima

Cover · Overview · Contents · Discuss · Suggest · Glossary · Index

2007

Pattern matching

Say you want to simplify arithmetic expressions of the kinds just presented. There is a multitude of possible simplification rules. The following three rules just serve as an illustration:

```
UnOp("-", UnOp("-", e)) => e // Double negation
BinOp("+", e, Number(0)) => e // Adding zero
BinOp("*", e, Number(1)) => e // Multiplying by one
```

Using **pattern matching**, these rules can be taken almost as they are to form the core of a simplification function in **Scala**, as shown in Listing 15.2. The function, **simplifyTop**, can be used like this:

```
scala> simplifyTop(UnOp("-", UnOp("-", Var("x"))))
res4: Expr = Var(x)
```

```
def simplifyTop(expr: Expr): Expr = expr match {
   case UnOp("-", UnOp("-", e)) => e // Double negation
   case BinOp("+", e, Number(0)) => e // Adding zero
   case BinOp("*", e, Number(1)) => e // Multiplying by one
   case _ => expr
}
```

Listing 15.2 · The **simplifyTop** function, which does a **pattern match**.

In the next two slides we go back to **Brian Goetz**'s blog post to see how **ADTs** for **ad-hoc** instances of **data structures** like **Option** and **Tree** can be written in **Java 19**.

@BrianGoetz

Application: Ad-hoc data structures

Algebraic data types are also useful for modeling **ad-hoc** versions of general purpose **data structures**. The popular class **Optional** could be modeled as an **algebraic data type**:

```
sealed interface Opt<T> {
  record Some<T>(T value) implements Opt<T> { }
  record None<T>() implements Opt<T> { }
}
```

(This is actually how **Optional** is defined in most functional languages.)

Common operations on **Opt** can be implemented with **pattern matching**:

```
static<T, U> Opt<U> map(Opt<T> opt, Function<T, U> mapper) {
  return switch (opt) {
    case Some<T>(var v) -> new Some<>(mapper.apply(v));
    case None<T>() -> new None<>();
  }
}
```

Similarly, a **binary tree** can be implemented as:

```
sealed interface Tree<T> {
  record Nil<T>() implements Tree<T> { }
  record Node<T>(Tree<T> left, T val, Tree<T> right) implements Tree<T> { }
}
```

@BrianGoetz

and we can implement the usual operations with pattern matching:

```
static<T> boolean contains(Tree<T> tree, T target) {
  return switch (tree) {
    case Nil() -> false;
    case Node(var left, var val, var right) ->
      target.equals(val) || left.contains(target) || right.contains(target);
    };
}

static<T> void inorder(Tree<T> t, Consumer<T> c) {
  switch (tree) {
    case Nil(): break;
    case Node(var left, var val, var right):
      inorder(left, c);
      c.accept(val);
      inorder(right, c);
  };
}
```

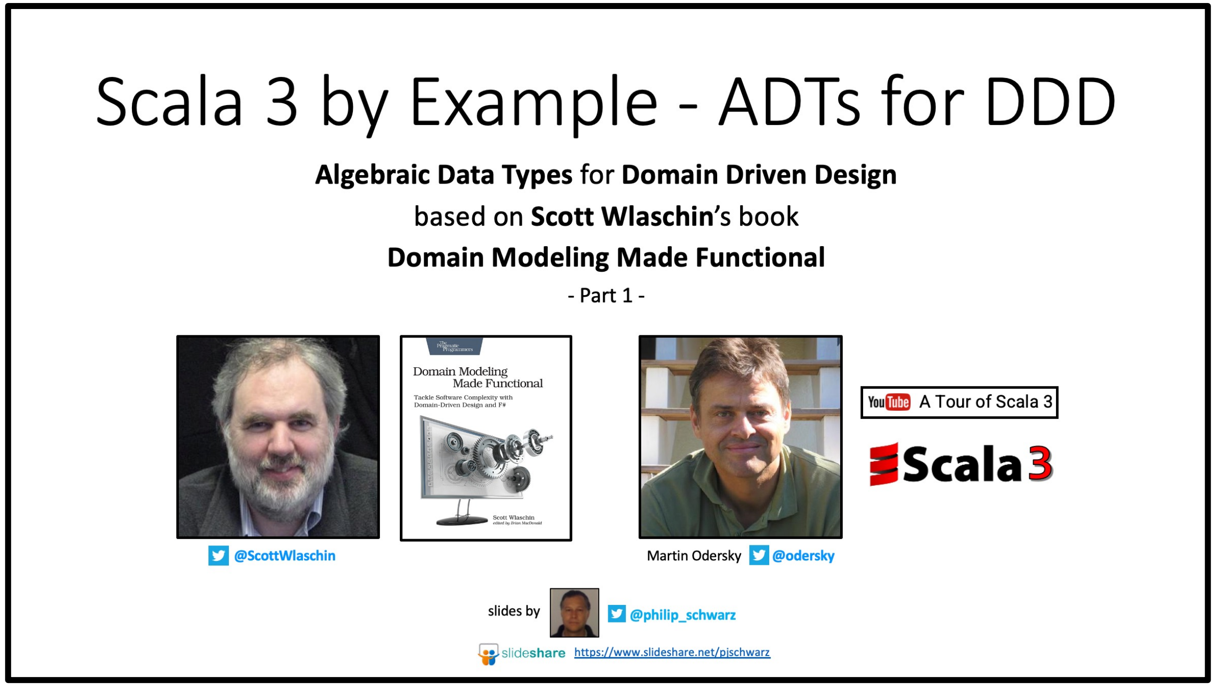
It may seem odd to see this behavior written as static methods, when common behaviors like traversal should "obviously" be implemented as abstract methods on the base interface. And certainly, some methods may well make sense to put into the interface. But **the combination of records, sealed classes, and pattern matching offers us alternatives that we didn't have before;** we could implement them the old fashioned way (with an abstract method in the base class and concrete methods in each subclass); as default methods in the abstract class implemented in one place with pattern matching; as static methods; or (when recursion is not needed), as ad-hoc traversals inline at the point of use.

Because the data carrier is purpose-built for the situation, **we get to choose whether we want the behavior to travel with the data or not**. **This approach is not at odds with object orientation; it is a useful addition to our toolbox that can be used alongside OO, as the situation demands.**

@philip_schwarz

While in **Programming in Scala** (first edition) we saw the three features that enable **Data Oriented programming**, we did not come across any references to the term **Algebraic Data Type**, so let us turn to later **Scala** books that do define the term.

By the way, if you are interested in a more comprehensive introduction to **Algebraic Data Types**, then take a look at the following deck, where the next four slides originate from:



Scala 3 by Example - ADTs for DDD

Algebraic Data Types for **Domain Driven Design**
based on **Scott Wlaschin**'s book
Domain Modeling Made Functional
- Part 1 -

@ScottWlaschin

Martin Odersky @odersky

A Tour of Scala 3

slides by @philip_schwarz

slideshare https://www.slideshare.net/pjschwarz

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

Defining functional data structures

A **functional data structure** is (not surprisingly) operated on using only pure functions. Remember, a pure function must not change data in place or perform other side effects. Therefore, **functional data structures are by definition immutable**.

…

let's examine what's **probably the most ubiquitous functional data structure, the singly linked list**. The definition here is identical in spirit to (though simpler than) the `List` data type defined in **Scala**'s standard library.

…

Let's look first at **the definition of the data type**, which **begins with the keywords `sealed trait`**.

In general, we introduce a data type with the `trait` keyword.

A `trait` is an **abstract interface** that may optionally contain implementations of some methods.

Here we're declaring a `trait`, called `List`, with **no methods** on it.

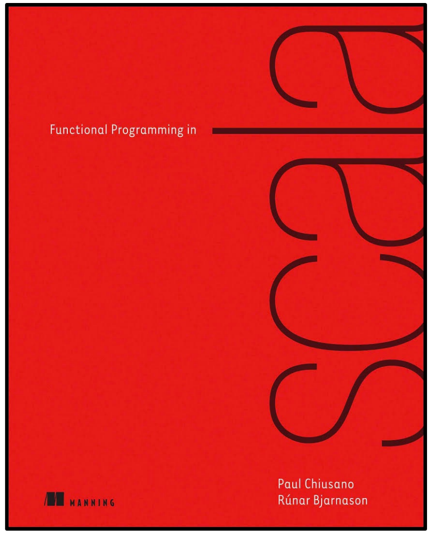
Adding `sealed` in front means that all implementations of the `trait` must be declared in this file.[1]

There are two such implementations, or **data constructors**, of `List` (each introduced with the keyword `case`) declared next, to represent the two possible forms a `List` can take.

As the figure…shows, a `List` can be empty, denoted by the data constructor `Nil`, or it can be nonempty, denoted by the data constructor `Cons` (traditionally short for construct). A nonempty list consists of an initial element, `head`, followed by a `List` (possibly empty) of remaining elements (the `tail`).

---

[1] We could also say `abstract class` here instead of `trait`. The distinction between the two is not at all significant for our purposes right now. …



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

3.5 Trees

`List` is just one example of what's called an **algebraic data type** (**ADT**). (Somewhat confusingly, **ADT** is sometimes used elsewhere to stand for **abstract data type**.) **An ADT is just a data type defined by one or more data constructors, each of which may contain zero or more arguments**. **We say that the data type is the sum or union of its data constructors, and each data constructor is the product of its arguments, hence the name algebraic data type**.[14]

[14] The naming is not coincidental. There's a **deep connection**, beyond the scope of this book, **between the "addition" and "multiplication" of types to form an ADT and addition and multiplication of numbers**.

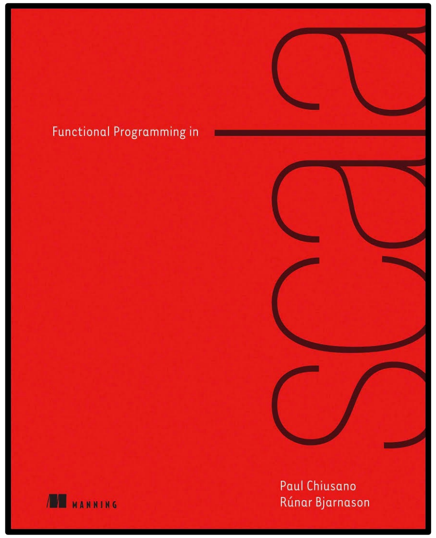
Tuple types in Scala

Pairs and tuples of other arities are also **algebraic data types**. They work just like the **ADT**s we've been writing here, but have special syntax…

Algebraic data types can be used to define other data structures. Let's define a simple binary tree data structure:

```
sealed trait Tree[+A]
case class Leaf[A](value: A) extends Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]
```

…



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

Let's recap (informally) what we just saw in **FPiS**.

- The **List algebraic data type** is the **sum of its data constructors**, **Nil** and **Cons**.
- The **Nil** constructor has no arguments.
- The **Cons** constructor is **the product of its arguments** `head: A` and `tail: List[A]`.

```scala
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

SUM

PRODUCT

- The **Tree algebraic data type** is the **sum of its data constructors**, **Leaf** and **Branch**.
- The **Leaf** constructor has a single argument.
- The **Branch** constructor is the **product of its arguments** `left: Tree[A]` and `right: Tree[A]`

```scala
sealed trait Tree[+A]
case class Leaf[A](value: A) extends Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]
```

SUM

PRODUCT

Algebraic Type Systems

Now we can define what we mean by an "**algebraic type system**." It's not as scary as it sounds—an **algebraic type system** is simply one where **every compound type is composed from smaller types by *AND*-ing or *OR*-ing them together. F#, like most functional languages (but unlike OO languages), has a built-in algebraic type system**.

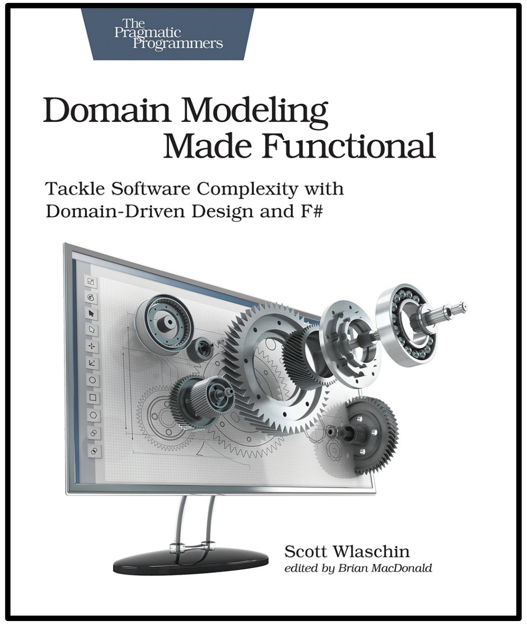
Using ***AND*** and ***OR*** to build new data types should feel familiar—we used the same kind of ***AND*** and ***OR*** to document our domain. We'll see shortly that **an algebraic type system is indeed an excellent tool for domain modeling**.

@ScottWlaschin

Jargon Alert: "Product Types" and "Sum Types"

The types that are built using ***AND*** are called ***product types***.

The types that are built using ***OR*** are called ***sum types*** or ***tagged unions*** or, in **F#** terminology, ***discriminated unions***. In this book **I will often call them *choice types*, because I think that best describes their role in domain modeling**.



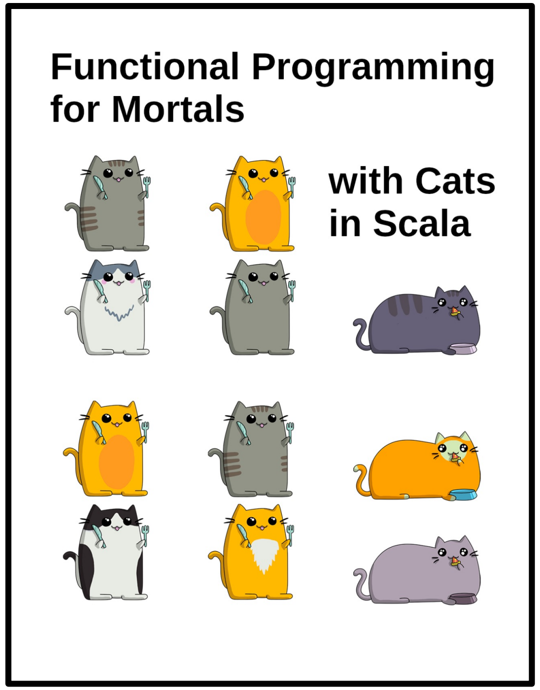
The Pragmatic Programmers

Domain Modeling Made Functional

Tackle Software Complexity with Domain-Driven Design and F#

Scott Wlaschin
edited by Brian MacDonald

In the next slide we see that **sum types**, as opposed to **product types**, are also known as **coproducts**.



Functional Programming for Mortals

with Cats in Scala

4.1 Data

The fundamental building blocks of data types are

- `final case class` also known as ***products***
- `sealed abstract class` also known as ***coproducts***
- `case object` and `Int`, `Double`, `String` (etc) ***values***

with no methods or fields other than the constructor parameters. We prefer `abstract class` to `trait` in order to get better binary compatibility and to discourage trait mixing. **The collective name for products, coproducts and values is Algebraic Data Type** (**ADT**).

We compose data types from the **AND** and **XOR** (exclusive OR) **Boolean algebra**: a **product** contains every type that it is composed of, but a **coproduct** can be only one. For example

- **product**: ABC = a **AND** b **AND** c
- **coproduct**: XYZ = x **XOR** y **XOR** z

written in **Scala**

```
// values
case object A
type B = String
type C = Int

// product
final case class ABC(a: A.type, b: B, c: C)

// coproduct
sealed abstract class XYZ
case object X extends XYZ
case object Y extends XYZ
```

4.1.1 Recursive ADTs

When an **ADT** refers to itself, we call it a **Recursive Algebraic Data Type**.

The standard library `List` is recursive because :: (the cons cell) contains a reference to `List`. The following is a simplification of the actual implementation:

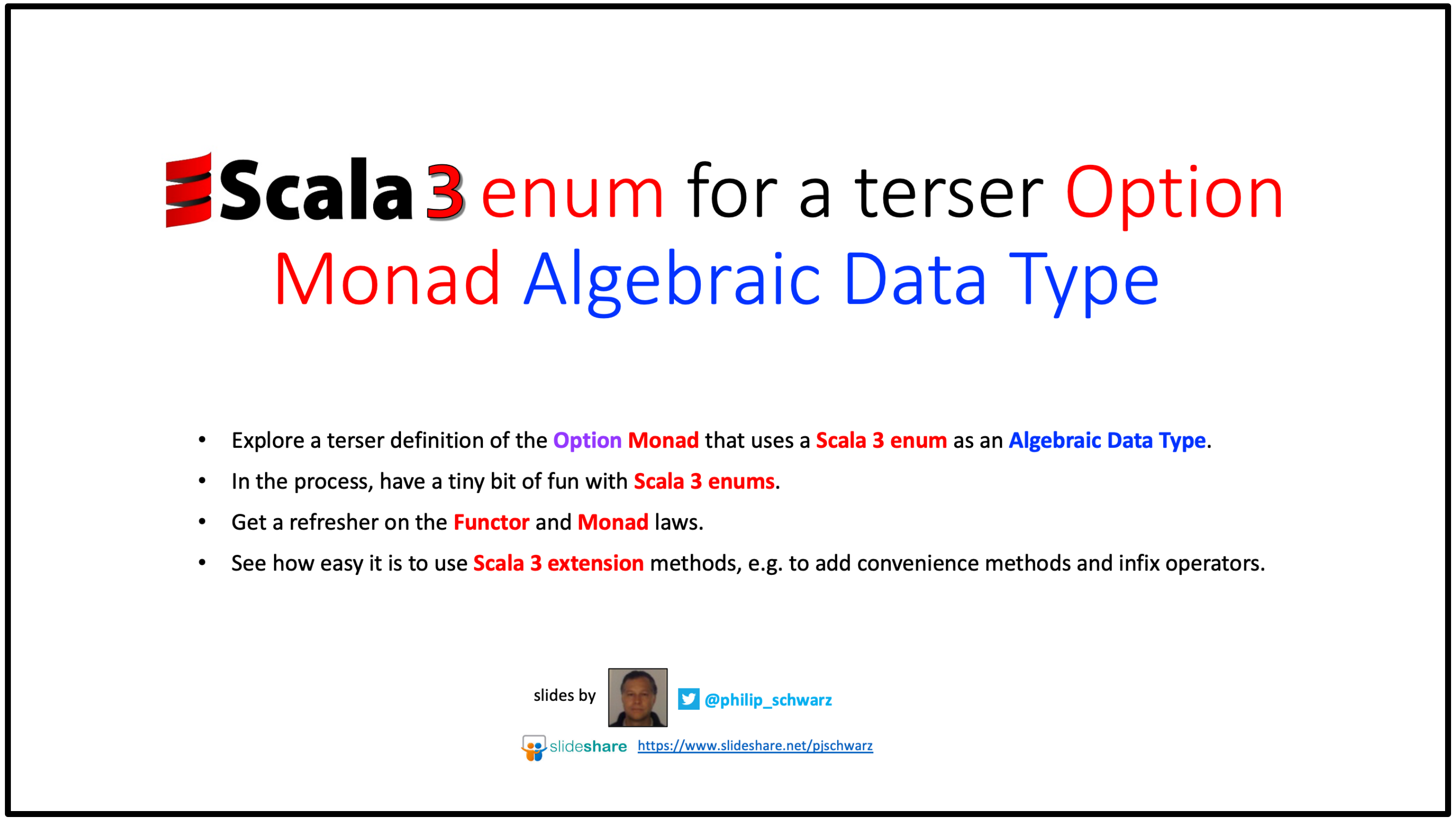
```
sealed abstract class List[+A]
case object Nil extends List[Nothing]
final case class ::[+A](head: A, tail: List[A]) extends List[A]
```

@philip_schwarz

In the next three slides, we'll see what **Brian Goetz** meant when he said, "that's how **Optional** is defined in most functional languages".

If you are not familiar with **Monads** then feel free to simply skim the third of those slides.

If you want to know more about the **Option Monad**, then see the following slide deck:



Scala 3 enum for a terser Option Monad Algebraic Data Type

- Explore a terser definition of the **Option Monad** that uses a **Scala 3 enum** as an **Algebraic Data Type**.
- In the process, have a tiny bit of fun with **Scala 3 enums**.
- Get a refresher on the **Functor** and **Monad** laws.
- See how easy it is to use **Scala 3 extension** methods, e.g. to add convenience methods and infix operators.

slides by **@philip_schwarz**

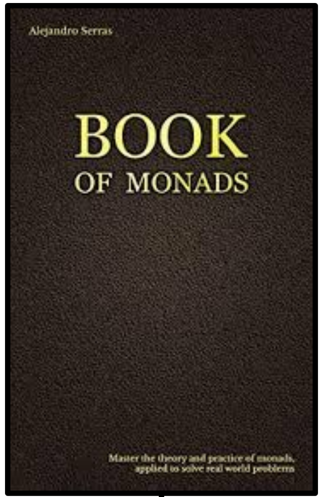
slideshare https://www.slideshare.net/pjschwarz

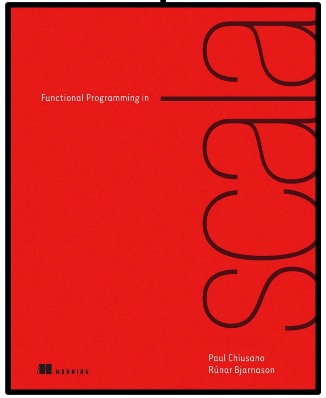
We introduce a new type, `Option`. As we mentioned earlier, this type also exists in the **Scala** standard library, but we're re-creating it here for pedagogical purposes:

```
sealed trait Option[+A]
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```

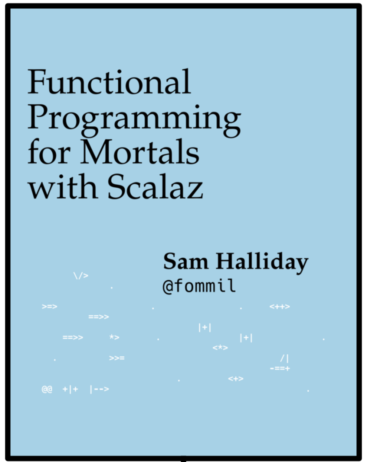
`Option` is mandatory! Do not use **null** to denote that an optional value is absent. Let's have a look at how `Option` is defined:

```
sealed abstract class Option[+A] extends IterableOnce[A]
final case class Some[+A](value: A) extends Option[A]
case object None extends Option[Nothing]
```





Over the years we have all got very used to the definition of the **Option monad**'s **Algebraic Data Type** (**ADT**).



Since creating new data types is so cheap, and it is possible to work with them polymorphically, most functional languages define some notion of an **optional value**. In **Haskell** it is called `Maybe`, in **Scala** it is `Option`, … Regardless of the language, the structure of the data type is similar:

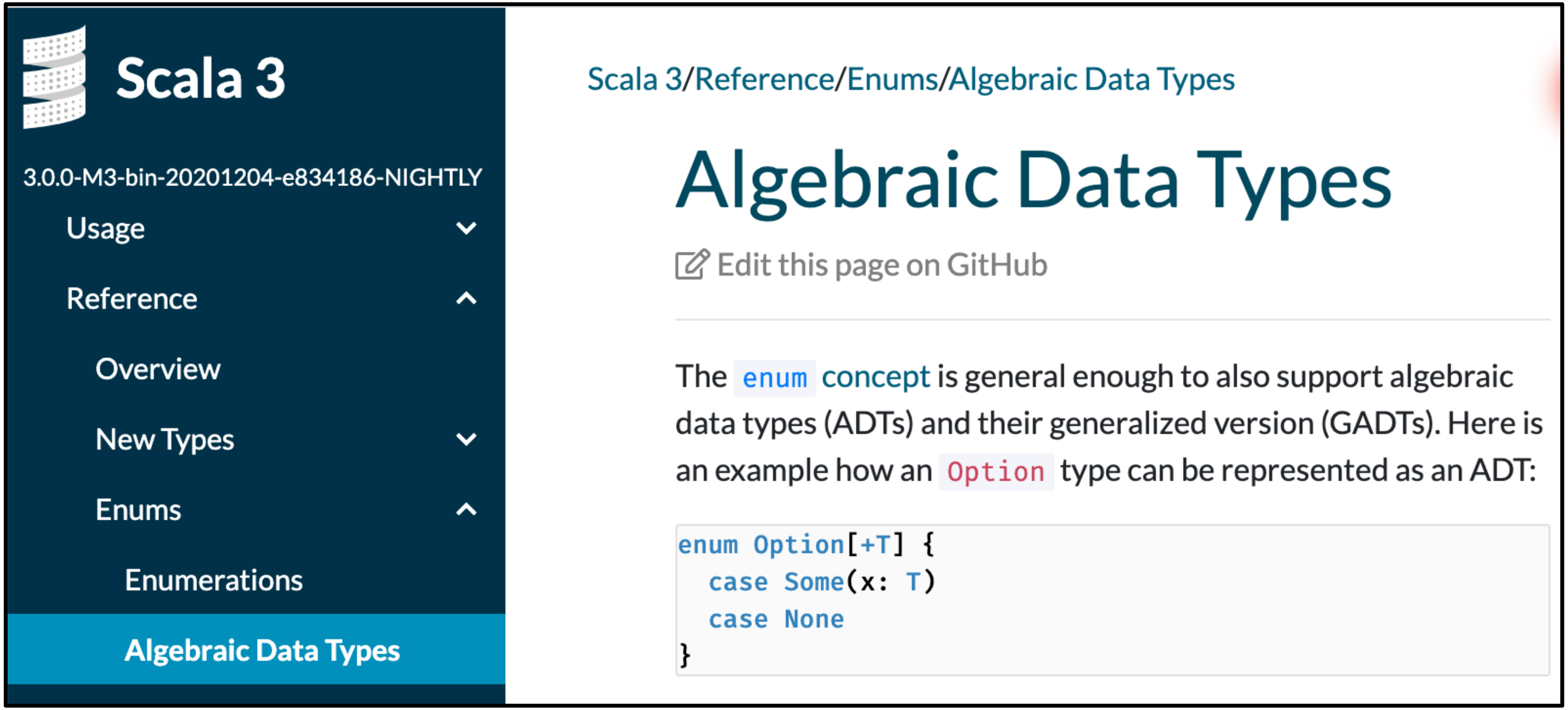
```
data Maybe a = Nothing -- no value
             | Just a  -- holds a value

sealed abstract class Option[+A]               // optional value
case object None extends Option[Nothing]       // no value
case class Some[A](value: A) extends Option[A] // holds a value
```

We have already encountered **scalaz**'s improvement over `scala.Option`, called **Maybe**. It is an improvement because it does not have any unsafe methods like `Option.get`, which can throw an exception, and is invariant.

It is typically used to represent when a thing may be present or not without giving any extra context as to why it may be missing.

```
sealed abstract class Maybe[A] { ... }
object Maybe {
  final case class Empty[A]() extends Maybe[A]
  final case class Just[A](a: A) extends Maybe[A]
```

With the arrival of **Scala 3** however, the definition of the **`Option`** **ADT** becomes much terser thanks to the fact that it can be implemented using the new **enum** concept .



Algebraic Data Types

The `enum` concept is general enough to also support algebraic data types (ADTs) and their generalized version (GADTs). Here is an example how an `Option` type can be represented as an ADT:

```
enum Option[+T] {
  case Some(x: T)
  case None
}
```

Option is a **monad**, so we have given it a **flatMap** method and a **pure** method. In **Scala** the latter is not strictly needed, but we'll make use of it later.

Every **monad** is also a **functor**, and this is reflected in the fact that we have given **Option** a **map** method.

We gave **Option** a **fold** method, to allow us to **interpret/execute** the **Option** **effect**, i.e. to escape from the **Option** container, or as **John a De Goes** puts it, to **translate away** from **optionality** by providing a **default value**.

We want our **Option** to integrate with **for comprehensions** sufficiently well for our current purposes, so in addition to **map** and **flatMap** methods, we have given it a simplistic **withFilter** method that is just implemented in terms of **filter**, another pretty essential method.

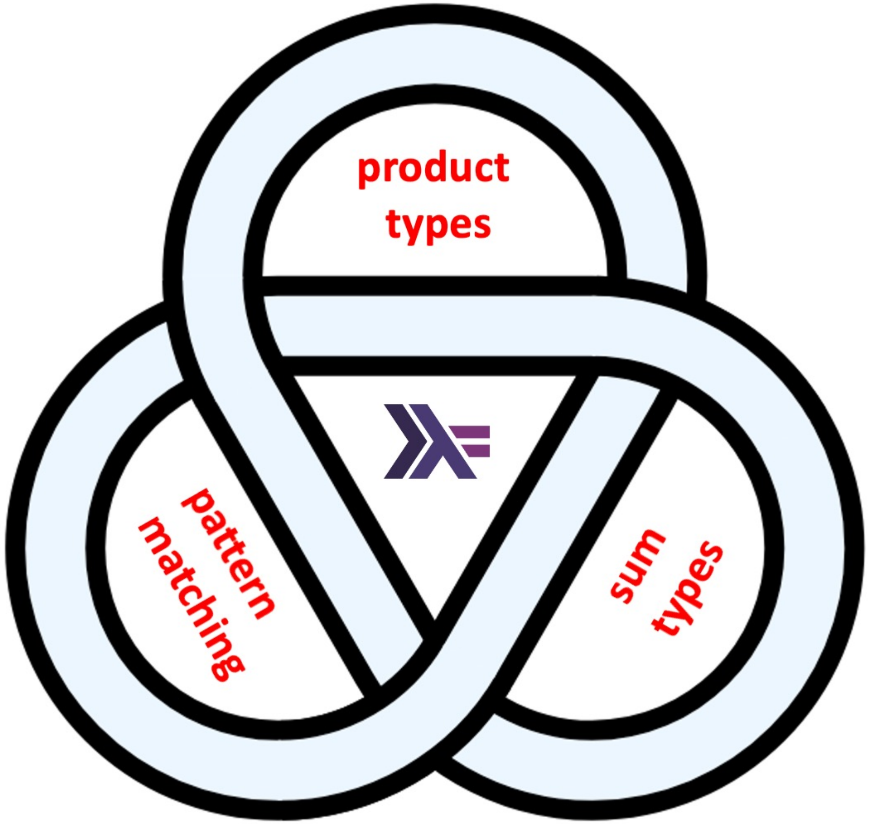
There are of course many many other methods that we would normally want to add to **Option**.

```
enum Option[+A]:
  case Some(a: A)
  case None

  def map[B](f: A => B): Option[B] =
    this match
      case Some(a) => Some(f(a))
      case None => None

  def flatMap[B](f: A => Option[B]): Option[B] =
    this match
      case Some(a) => f(a)
      case None => None

  def fold[B](ifEmpty: => B)(f: A => B) =
    this match
      case Some(a) => f(a)
      case None => ifEmpty

  def filter(p: A => Boolean): Option[A] =
    this match
      case Some(a) if p(a) => Some(a)
      case _ => None

  def withFilter(p: A => Boolean): Option[A] =
    filter(p)

object Option :
  def pure[A](a: A): Option[A] = Some(a)
  def none: Option[Nothing] = None

extension[A](a: A):
  def some: Option[A] = Some(a)
```

The **some** and **none** methods are just there to provide the convenience of **Cats**-like syntax for **lifting** a **pure** value into an **Option** and for referring to the **empty Option** instance.

What about **Algebraic Data Types** in **Haskell**?

Let’s turn to that in the next four slides.



Algebraic types and pattern matching

Algebraic data types can express a **combination** of **types**, for example:

```
type Name = String
type Age = Int
data Person = P String Int -- combination
```

They can also express a **composite** of **alternatives**:

```
data MaybeInt = NoInt | JustInt Int
```

Here, each **alternative** represents a valid **constructor** of the **algebraic type**:

```
maybeInts = [JustInt 2, JustInt 3, JustInt 5, NoInt]
```

Type combination is also known as “**product of types**” and the **type alternation** as “**sum of types**”. In this way, we can create an “**algebra of types**”, with **sum** and **product** as **operators**, hence the name **Algebraic data types**.

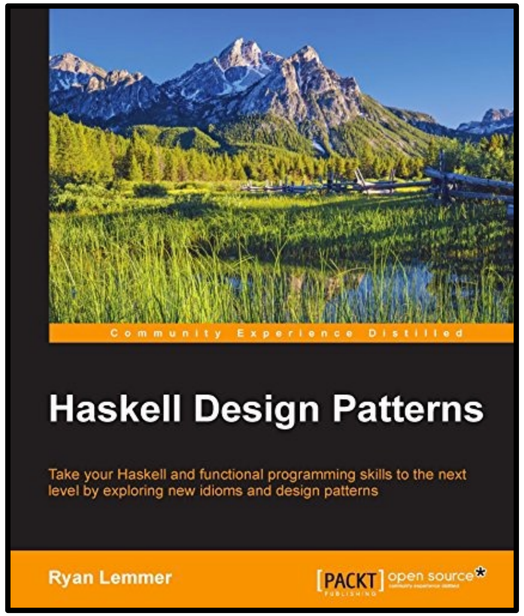
By parametrizing algebraic types, we can create generic types:

```
data Maybe' a = Nothing' | Just’ a
```

Algebraic data type constructors also serve as “**deconstructors**“ in **pattern matching**:

```
fMaybe f (Just' x) = Just' (f x)
fMaybe f Nothing' = Nothing’

fMaybes = map (fMaybe (* 2)) [Just’ 2, Just’ 3, Nothing]
```

On the left of the = sign we **deconstruct**; on the right we **construct**. In this sense, **pattern matching is the complement of algebraic data types: they are two sides of the same coin**.



16.1 Product types—combining types with "and"

Product types are created by **combining** two or more existing types with *and*. Here are some common examples:

- A fraction can be defined as a numerator (`Integer`) *and* denominator (another `Integer`).
- A street address might be a number (`Int`) *and* a street name (`String`).
- A mailing address might be a street address *and* a city (`String`) *and* a state (`String`) *and* a zip code (`Int`).

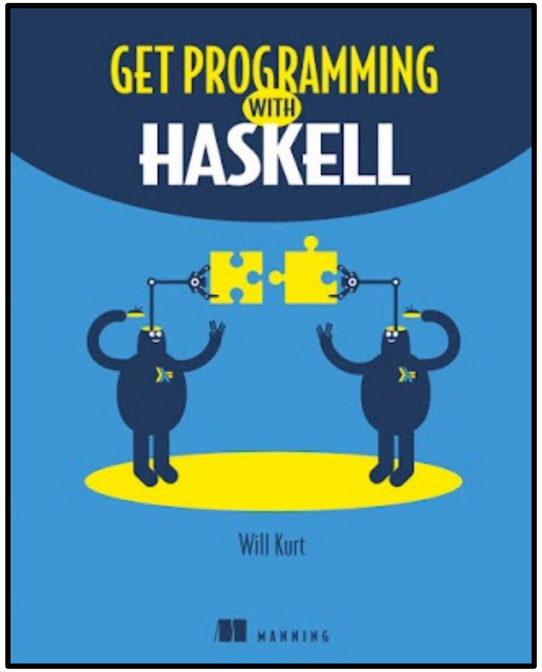
Although the name *product type* might make this method of combining types sound sophisticated, this is the most common way in all programming languages to define types. Nearly all programming languages support product types. The simplest example is a struct from C. Here's an example in C of a struct for a book and an author.

Listing 16.1 C structs are product types—an example with a book and author

```
struct author_name {
  char *first_name;
  char *last_name;
};

struct book {
  author_name author;
  char *isbn;
  char *title;
  int year_published;
  double price;
};
```

Listing 16.2 C's author_name and book structs translated to Haskell

```
data AuthorName = AuthorName String String
data Book = Author String String Int
```



In this example, you can see that the author_name type is made by **combining** two `Strings` (for those unfamiliar, char * in C represents an array of characters). The book type is made by **combining** an author_name, two `Strings`, an `Int`, and a `Double`. Both author_name and book are made by **combining** other types with an *and*. C's structs are the predecessor to similar types in nearly every language, including classes and JSON.

16.2 Sum types—combining types with “or ”

Sum types are a surprisingly powerful tool, given that they provide only the capability to **combine** two types with *or*. Here are examples of **combining types** with *or*:

- A die is **either** a 6-sided die **or** a 20-sided die or ....
- A paper is authored by **either** a person (`String`) **or** a group of people (`[String]`).
- A list is **either** an empty list (`[]`) **or** an item consed with another list (`a:[a]`).

The most straightforward **sum type** is `Bool`.

Listing 16.8 A common sum type: Bool

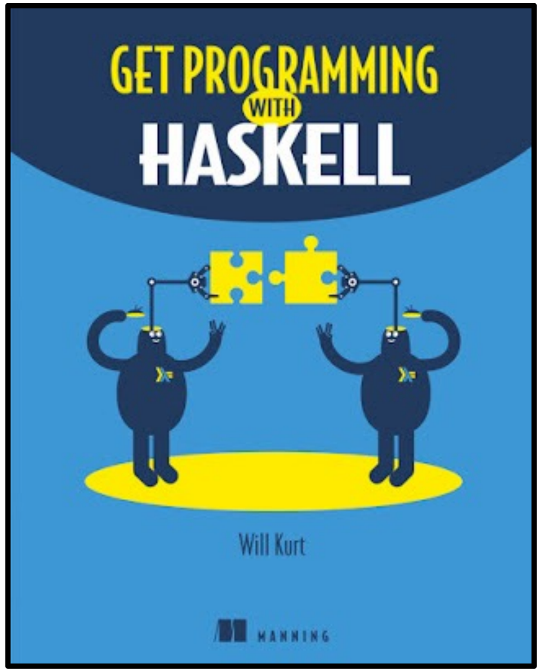
```
data Bool = False | True
```

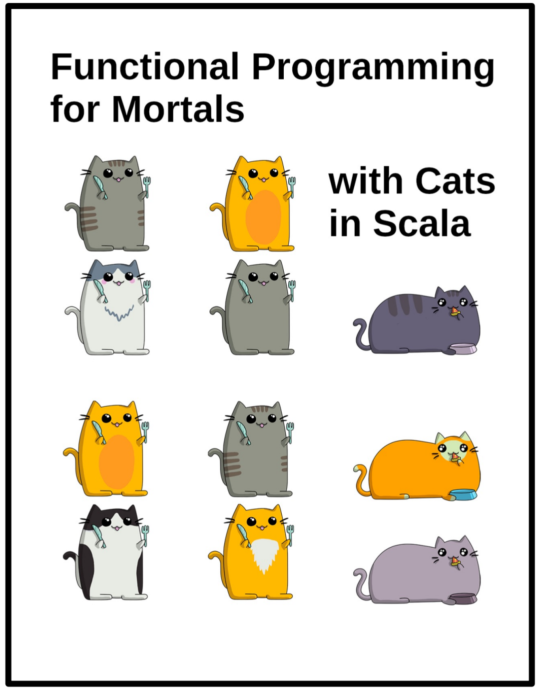
An instance of `Bool` is either the `False` data constructor or the `True` data constructor. This can give the mistaken impression that sum types are just Haskell’s way of creating enu- merative types that exist in many other programming languages. But you’ve already seen a case in which sum types can be used for something more powerful, in lesson 12 when you defined two types of names.

Listing 16.9 Using a sum type to model names with and without middle names

```
type FirstName = String
type LastName = String
type MiddleName = String

data Name = Name FirstName LastName | NameWithMiddle FirstName MiddleName LastName
```

In this example, you can use two type constructors that can either be a `FirstName` consisting of two `Strings` *or* a `NameWithMiddle` consisting of three `Strings`. Here, using *or* between two types allows you to be expressive about what types mean. **Adding *or* to the tools you can use to combine types opens up worlds of possibility in Haskell that aren’t available in any other programming language without sum types**.





Functional Programming for Mortals

with Cats in Scala

Data

Haskell has a very clean syntax for **ADTs**. This is a **linked list** structure:

```
data List a = Nil | Cons a (List a)
```

`List` is a **type constructor**, a is the type parameter, | separates the **data constructors**, which are: `Nil` the **empty list** and a `Cons` cell. `Cons` takes two parameters, which are separated by whitespace: no commas and no parameter brackets.

There is no **subtyping** in **Haskell**, so there is no such thing as the `Nil` type or the `Cons` type: both construct a `List`.

In his blog post, **Brian Goetz** first looked at the following sample applications of **Data Oriented programming**:

- Complex return types (we skipped this)
- Ad-hoc data structures (we covered this)

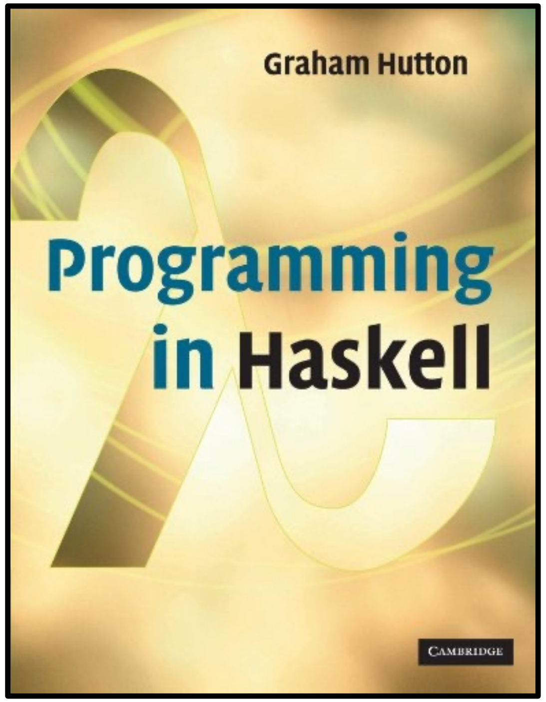
He then turned to more complex domains and chose as an example the **evaluation** of simple **arithmetic expressions**.

This is a **classic example** of using **ADTs**.

We got a first hint of the **expression ADT** (albeit a slightly more complex version) in the first edition of **Programming in Scala**:

```
sealed abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr) extends Expr
```

See the next two slides for when I first came across examples of the expression **ADT** in **Haskell** and **Scala**.



2007

8.7 Abstract machine

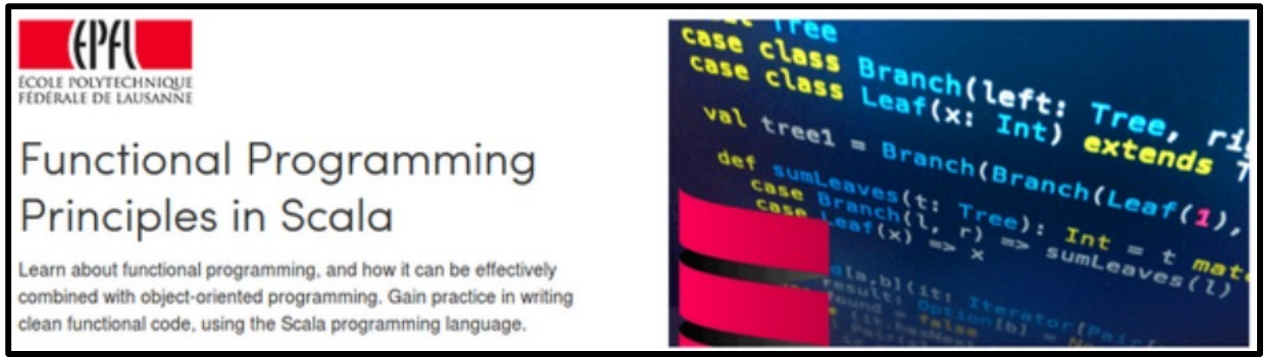
For our second extended example, consider a type of simple **arithmetic expressions** built up from integers using an addition operator, together with a function that evaluates such an **expression** to an integer value:

```
data Expr = Val Int | Add Expr Expr

value :: Expr -> Int
value (Val n) = n
value (Add x y) = value x + value y
```

For example, the expression (2 + 3) + 4 is evaluated as follows:

```
    value (Add (Add (Val 2) (Val 3)) (Val 4))
= { applying value }
    value (Add (Val 2) (Val 3)) + value (Val 4)
= { applying the first value }
    (value (Val 2) + value (Val 3)) + value (Val 4)
= { applying the first value }
    (2 + value (Val 3)) + value (Val 4)
= { applying the first value }
    (2 + 3) + value (Val 4)
= { applying the first + }
    5 + value (Val 4)
= { applying value }
    5 + 4
= { applying + }
    9
```

In did this course in 2013 (the second edition more recently). The lectures for the first edition are freely available on YouTube.



ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

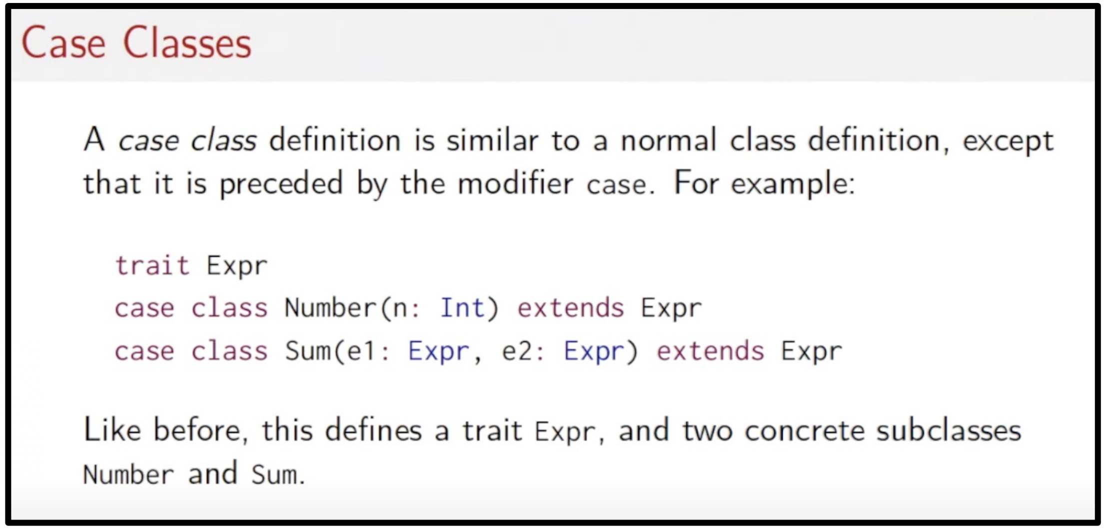
Functional Programming Principles in Scala

Learn about functional programming, and how it can be effectively combined with object-oriented programming. Gain practice in writing clean functional code, using the Scala programming language.

Functional Programming Principles in Scala

1.18K subscribers

4.7 Pattern Matching

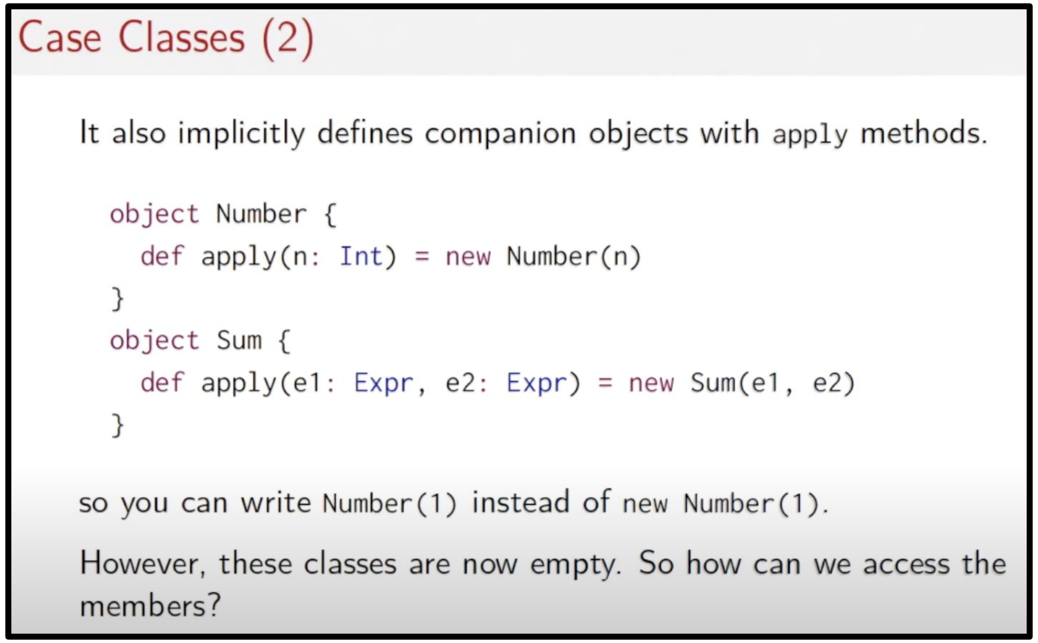


Case Classes

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`.



Case Classes (2)

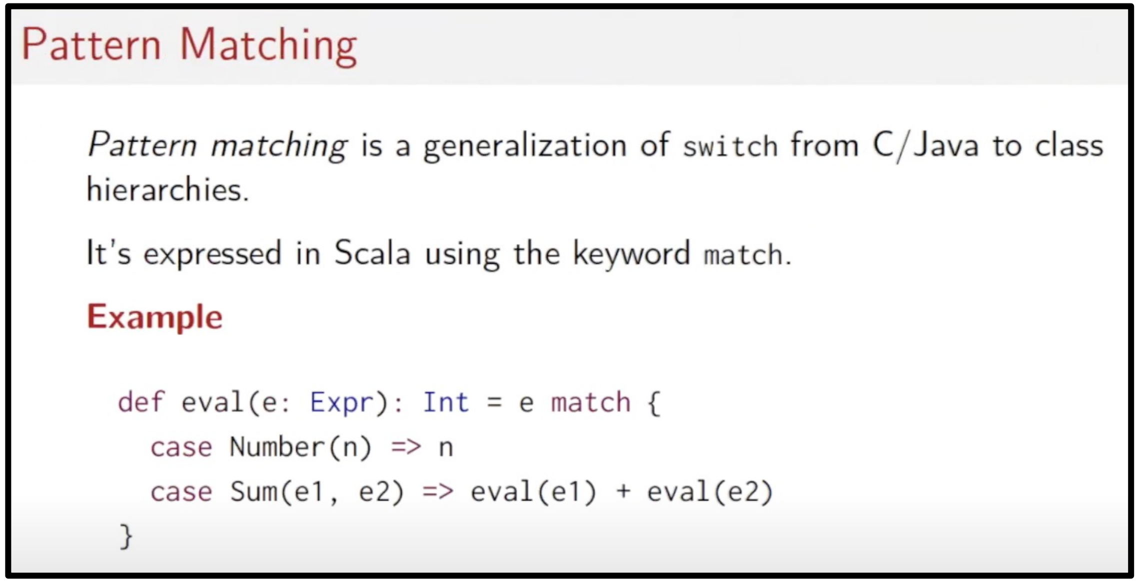
It also implicitly defines companion objects with `apply` methods.

```
object Number {
  def apply(n: Int) = new Number(n)
}
object Sum {
  def apply(e1: Expr, e2: Expr) = new Sum(e1, e2)
}
```

so you can write `Number(1)` instead of `new Number(1)`.

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

Martin Odersky



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)
}
```

Next, let's see **Brian Goetz** present his **expression ADT**.

@philip_schwarz

@BrianGoetz

More complex domains

The domains we've looked at so far have either been "**throwaways**" (return values used across a call boundary) or modeling general domains like **lists** and **trees**. But the same approach is also useful for more complex **application-specific domains**. If we wanted to model an **arithmetic expression**, we could do so with:

```
sealed interface Node { }
sealed interface BinaryNode extends Node {
  Node left(); Node right();
}

record AddNode(Node left, Node right) implements BinaryNode { }
record MulNode(Node left, Node right) implements BinaryNode { }
record ExpNode(Node left, int exp) implements Node { }
record NegNode(Node node) implements Node { }
record ConstNode(double val) implements Node { }
record VarNode(String name) implements Node { }
```

Having the intermediate **sealed interface** `BinaryNode` which abstracts over addition and multiplication gives us the choice when matching over a `Node`; we could handle both addition and multiplication together by matching on `BinaryNode`, or handle them individually, as the situation requires. The language will still make sure we covered all the cases.

@BrianGoetz

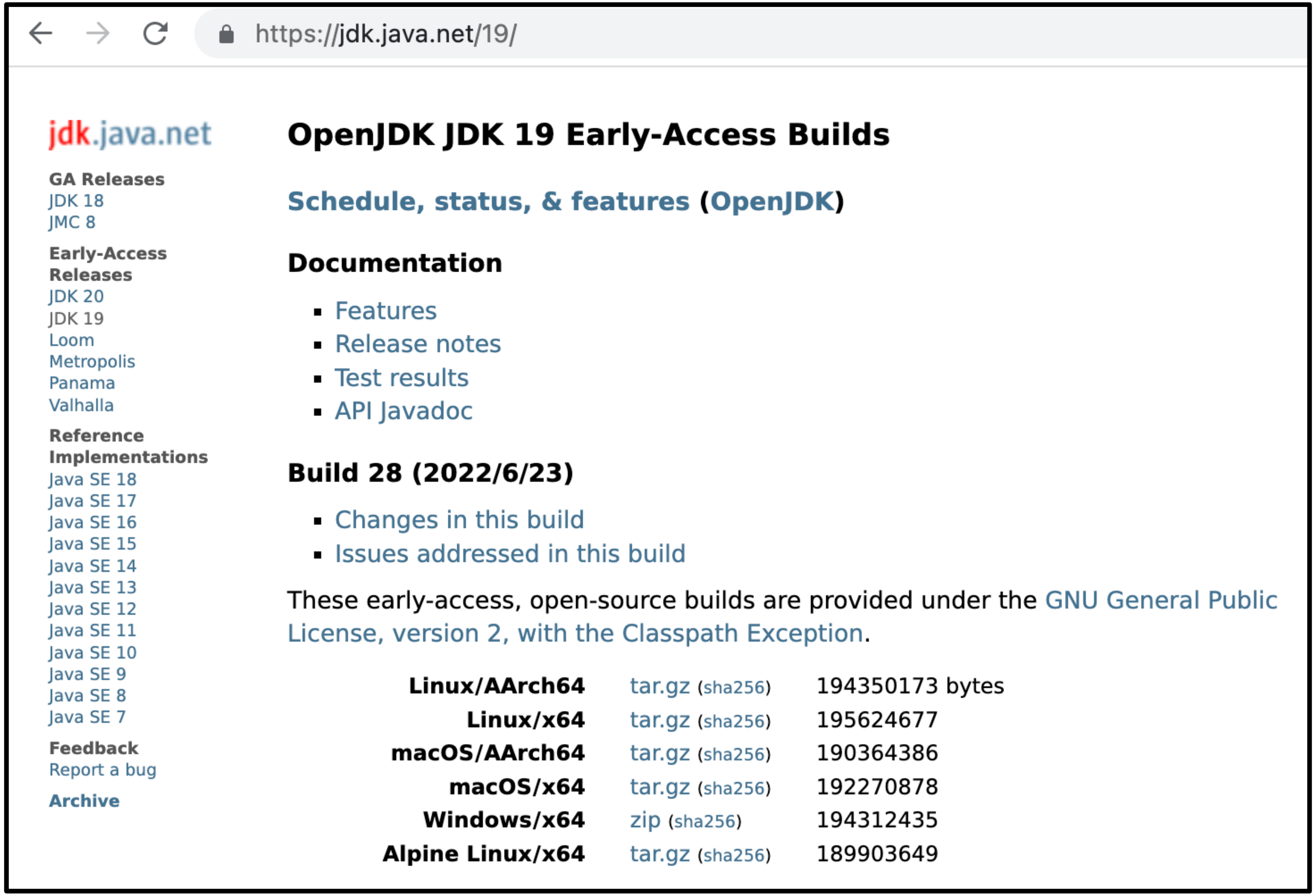
Writing an evaluator for these expressions is trivial. Since we have variables in our expressions, we'll need a store for those, which we pass into the evaluator:

```
double eval(Node n, Function<String, Double> vars) {
  return switch (n) {
    case AddNode(var left, var right) -> eval(left, vars) + eval(right, vars);
    case MulNode(var left, var right) -> eval(left, vars) * eval(right, vars);
    case ExpNode(var node, int exp) -> Math.exp(eval(node, vars), exp);
    case NegNode(var node) -> -eval(node, vars);
    case ConstNode(double val) -> val;
    case VarNode(String name) -> vars.apply(name);
  }
}
```

The records which define the terminal nodes have reasonable toString implementations, but the output is probably more verbose than we'd like. We can easily write a formatter to produce output that looks more like a mathematical expression:

```
String format(Node n) {
  return switch (n) {
    case AddNode(var left, var right) -> String.format("("%s + %s)", format(left), format(right));
    case MulNode(var left, var right) -> String.format("("%s * %s)", format(left), format(right));
    case ExpNode(var node, int exp) -> String.format("%s^%d", format(node), exp);
    case NegNode(var node) -> String.format("-%s", format(node));
    case ConstNode(double val) -> Double.toString(val);
    case VarNode(String name) -> name;
  }
}
```

In order to run that code, I downloaded the **Java 19** early access build.



GA Releases
- JDK 18
- JMC 8

Early-Access Releases
- JDK 20
- JDK 19
- Loom
- Metropolis
- Panama
- Valhalla

Reference Implementations
- Java SE 18
- Java SE 17
- Java SE 16
- Java SE 15
- Java SE 14
- Java SE 13
- Java SE 12
- Java SE 11
- Java SE 10
- Java SE 9
- Java SE 8
- Java SE 7

Feedback
- Report a bug

Archive

OpenJDK JDK 19 Early-Access Builds

Schedule, status, & features (OpenJDK)

Documentation

- Features
- Release notes
- Test results
- API Javadoc

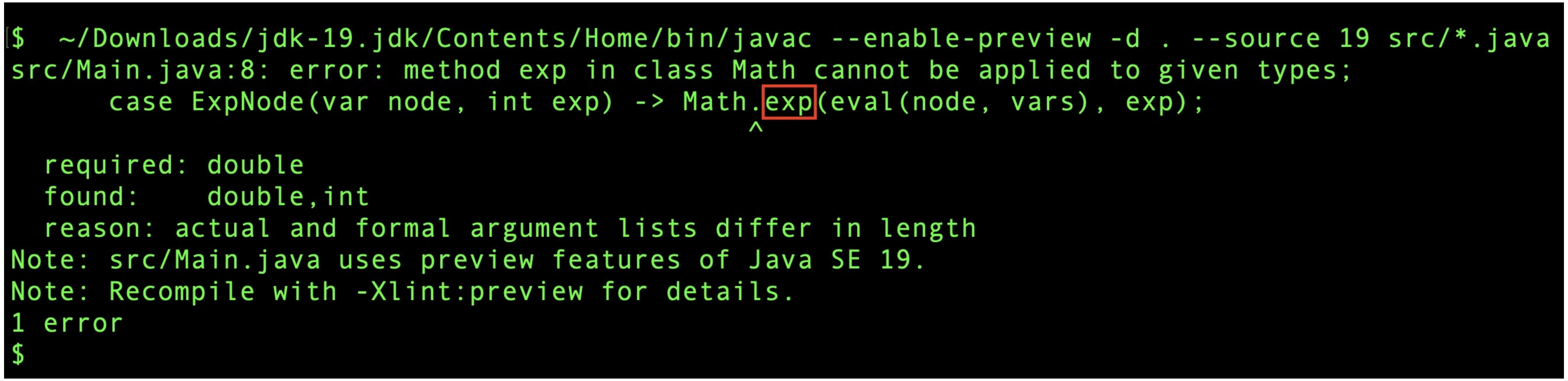
Build 28 (2022/6/23)

- Changes in this build
- Issues addressed in this build

These early-access, open-source builds are provided under the GNU General Public License, version 2, with the Classpath Exception.

Linux/AArch64	tar.gz (sha256)	194350173 bytes
Linux/x64	tar.gz (sha256)	195624677
macOS/AArch64	tar.gz (sha256)	190364386
macOS/x64	tar.gz (sha256)	192270878
Windows/x64	zip (sha256)	194312435
Alpine Linux/x64	tar.gz (sha256)	189903649

When I tried to compile the code, I got the following error, so I replaced the call to **Math.exp** with a call to **Math.pow** and renamed **ExpNode** to **PowNode**.



```
[$  ~/Downloads/jdk-19.jdk/Contents/Home/bin/javac --enable-preview -d . --source 19 src/*.java
src/Main.java:8: error: method exp in class Math cannot be applied to given types;
      case ExpNode(var node, int exp) -> Math.exp(eval(node, vars), exp);
                                             ^

  required: double
  found:    double,int
  reason: actual and formal argument lists differ in length
Note: src/Main.java uses preview features of Java SE 19.
Note: Recompile with -Xlint:preview for details.
1 error
$
```

For the sake of consistency with the classic **expression ADT**, I also did the following:

- renamed **Node** to **`Expr`**
- renamed **`AddNode`**, **`MulNode`**, etc. to **`Add`**, **`Mul`**, etc.
- dropped the **`BinaryNode`** interface

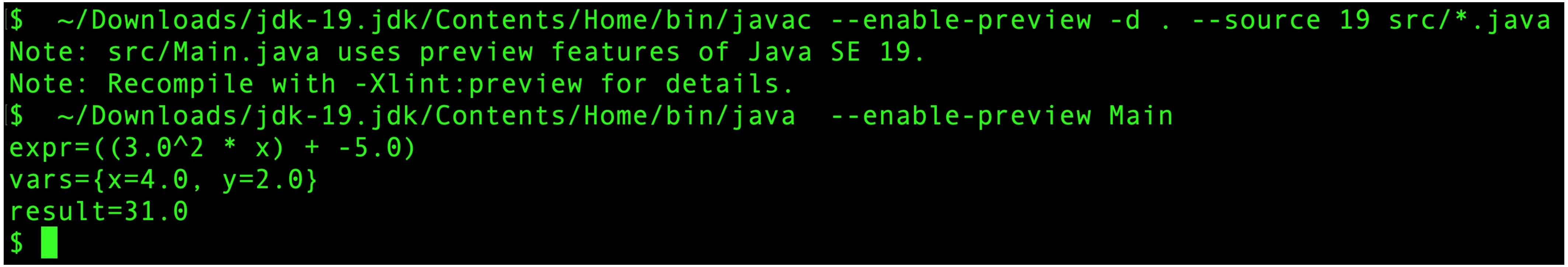
See next slide for the resulting code.

```java
import java.util.Map;
import java.util.function.Function;

public class Main {

  static double eval(Expr e, Function<String, Double> vars) {
    return switch (e) {
      case Add(var left, var right) -> eval(left, vars) + eval(right, vars);
      case Mul(var left, var right) -> eval(left, vars) * eval(right, vars);
      case Pow(var expr, int exp) -> Math.pow(eval(expr, vars), exp);
      case Neg(var expr) -> -eval(expr, vars);
      case Const(double val) -> val;
      case Var(String name) -> vars.apply(name);
    };
  }

  static String format(Expr e) {
    return switch (e) {
      case Add(var left, var right) -> String.format("(%s + %s)", format(left), format(right));
      case Mul(var left, var right) -> String.format("(%s * %s)", format(left), format(right));
      case Pow(var expr, int exp) -> String.format("%s^%d", format(expr), exp);
      case Neg(var expr) -> String.format("-%s", format(expr));
      case Const(double val) -> Double.toString(val);
      case Var(String name) -> name;
    };
  }

  static Map<String,Double> bindings = Map.of("x",4.0,"y", 2.0);

  static Function<String,Double> vars = v -> bindings.getOrDefault(v, 0.0);

  public static void main(String[] args) { … }

}
```

```java
public sealed interface Expr { }
record Add(Expr left, Expr right) implements Expr { }
record Mul(Expr left, Expr right) implements Expr { }
record Pow(Expr left, int exp) implements Expr { }
record Neg(Expr expr) implements Expr { }
record Const(double val) implements Expr { }
record Var(String name) implements Expr { }
```

```java
public static void main(String[] args) {

   var expr =
       new Add(
           new Mul(
               new Pow(new Const(3.0),2),
               new Var("x")),
           new Neg(new Const(5.0)));

   System.out.println(”expr=” + format(expr));
   System.out.println("vars=” + bindings);
   System.out.println(”value=” + eval(expr,vars));
 }
```

Let’s run that code:



```
[$  ~/Downloads/jdk-19.jdk/Contents/Home/bin/javac --enable-preview -d . --source 19 src/*.java
Note: src/Main.java uses preview features of Java SE 19.
Note: Recompile with -Xlint:preview for details.
[$  ~/Downloads/jdk-19.jdk/Contents/Home/bin/java  --enable-preview Main
expr=((3.0^2 * x) + -5.0)
vars={x=4.0, y=2.0}
result=31.0
$ 
```

Now let's translate that code into **Scala**.

```scala
def eval(e: Expr, vars: String => Double): Double =
  e match
    case Add(left, right) => eval(left, vars) + eval(right, vars)
    case Mul(left, right) => eval(left, vars) * eval(right, vars)
    case Pow(expr, exp) => Math.pow(eval(expr, vars), exp)
    case Neg(expr) => - eval(expr, vars)
    case Const(value) => value
    case Var(name) => vars(name)

def format(e: Expr): String =
  e match
    case Add(left, right) => s"(${format(left)} + ${format(right)})"
    case Mul(left, right) => s"(${format(left)} * ${format(right)})"
    case Pow(expr, exp) => s"${format(expr)}^$exp"
    case Neg(expr) => s"-${format(expr)}"
    case Const(value) => value.toString
    case Var(name) => name

val bindings = Map( "x" -> 4.0, "y" -> 2.0)

def vars(v: String): Double = bindings.getOrElse(v, 0.0)

@main def main(): Unit =
  val expr = Add(
               Mul(
                 Pow(Const(3.0),2),
                 Var("x")),
               Neg(Const(5.0)))

  println(s"expr=${format(expr)}")
  println(s"vars=$bindings")
  println(s"result=${eval(expr,vars)}")
```

```scala
enum Expr:
  case Add(left: Expr, right: Expr)
  case Mul(left: Expr, right: Expr)
  case Pow(left: Expr, exp: Int)
  case Neg(node: Expr)
  case Const(value: Double)
  case Var(name: String)
```

Scala

```
expr=((3.0^2 * x) + -5.0)
vars=Map(x -> 4.0, y -> 2.0)
result=31.0
```

```
def eval(e: Expr, vars: String => Double): Double =
  e match
    case Add(left, right) => eval(left, vars) + eval(right, vars)
    case Mul(left, right) => eval(left, vars) * eval(right, vars)
    case Pow(expr, exp) => Math.pow(eval(expr, vars), exp)
    case Neg(expr) => - eval(expr, vars)
    case Const(value) => value
    case Var(name) => vars(name)

def format(e: Expr): String =
  e match
    case Add(left, right) => s"(${format(left)} + ${format(right)})"
    case Mul(left, right) => s"(${format(left)} * ${format(right)})"
    case Pow(expr, exp) => s"${format(expr)}^$exp"
    case Neg(expr) => s"-${format(expr)}"
    case Const(value) => value.toString
    case Var(name) => name

val bindings = Map( "x" -> 4.0, "y" -> 2.0)

def vars(v: String): Double = bindings.getOrElse(v, 0.0)

@main def main(): Unit =
  val expr = Add(
               Mul(
                 Pow(Const(3.0),2),
                 Var("x")),
               Neg(Const(5.0)))

  println(s"expr=${format(expr)}")
  println(s"vars=$bindings")
  println(s"result=${eval(expr,vars)}")
```

```
sealed trait Expr
case class Add(left: Expr, right: Expr) extends Expr
case class Mul(left: Expr, right: Expr) extends Expr
case class Pow(expr: Expr, exp: Int) extends Expr
case class Neg(expr: Expr) extends Expr
case class Const(value: Double) extends Expr
case class Var(name: String) extends Expr
```

Scala

Same code as on the previous slide, except that for the **ADT**, instead of using the **syntactic sugar** afforded by **enum**, we use the more verbose **sealed trait** plus **case classes**.

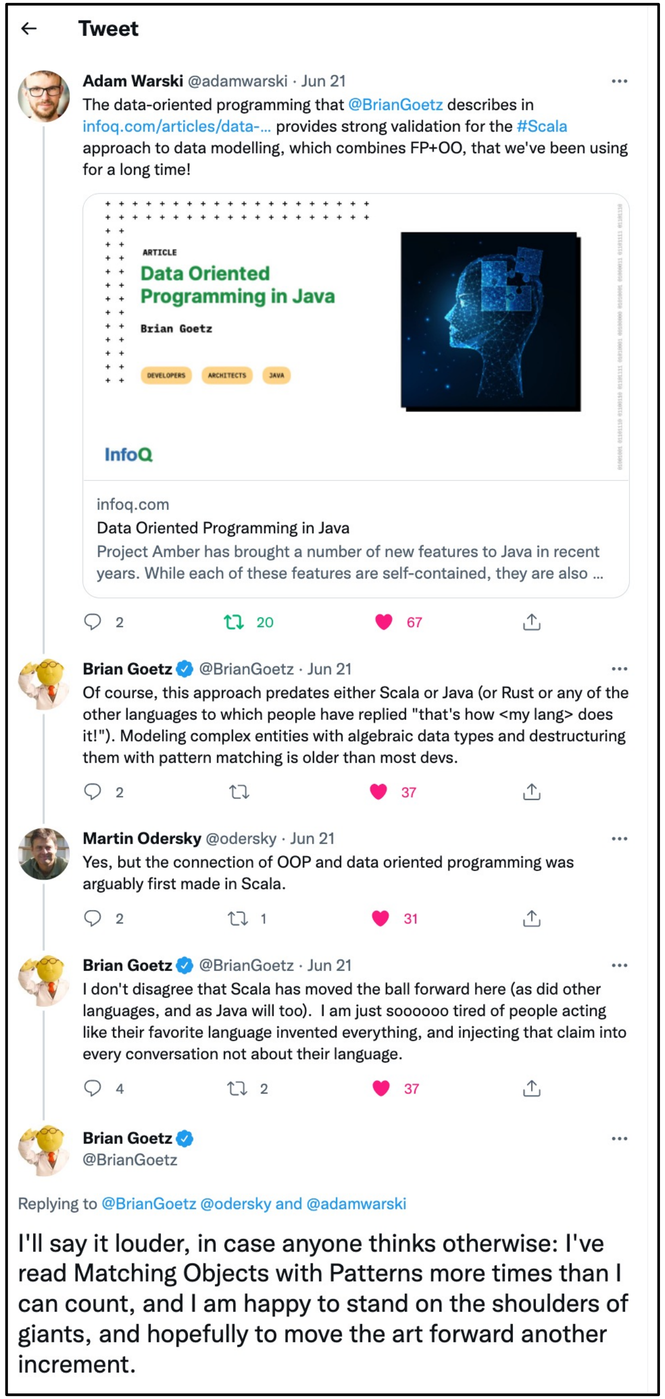
And now, to conclude this slide deck, let’s translate the code into **Haskell**.

```haskell
data Expr = Add Expr Expr |
            Mul Expr Expr |
            Pow Expr Int  |
            Neg Expr      |
            Const Double  |
            Var String

eval :: Expr -> (String -> Double) -> Double
eval (Add l r) vars = eval l vars + eval r vars
eval (Mul l r) vars = eval l vars * eval r vars
eval (Pow e n) vars = eval e vars ^ n
eval (Neg e) vars = - eval e vars
eval (Const i) _ = i
eval (Var v) vars = vars v

format :: Expr -> String
format (Add l r) = "(" ++ format l ++ " + " ++ format r ++ ")"
format (Mul l r) = "(" ++ format l ++ " * " ++ format r ++ ")"
format (Pow e n) = format e ++ " ^ " ++ show n
format (Neg e) = "-" ++ format e
format (Const i) = show i
format (Var v) = v

bindings =  [("x", 4.0), ("y", 2.0)]

vars :: String -> Double
vars v = maybe undefined id (lookup v bindings)
```

```haskell
main :: IO ()
main = let expression = (Add
                          (Mul
                            (Pow (Const 3.0) 2)
                            (Var "x")
                          )
                          (Neg (Const 5.0))
                        )
  in do putStrLn ("expr=" ++ format expression)
        putStr "vars="
        print bindings
        putStrLn ("value=" ++ show (eval expression vars))
```

Haskell

```
expr=((3.0 ^ 2 * x) + -5.0)
vars=[("x",4.0),("y",2.0)]
value=31.0
```



Adam Warski @adamwarski · Jun 21

The data-oriented programming that @BrianGoetz describes in infoq.com/articles/data-... provides strong validation for the #Scala approach to data modelling, which combines FP+OO, that we've been using for a long time!

ARTICLE

Data Oriented Programming in Java

Brian Goetz

DEVELOPERS ARCHITECTS JAVA

InfoQ

infoq.com

Data Oriented Programming in Java

Project Amber has brought a number of new features to Java in recent years. While each of these features are self-contained, they are also ...

Brian Goetz @BrianGoetz · Jun 21

Of course, this approach predates either Scala or Java (or Rust or any of the other languages to which people have replied "that's how <my lang> does it!"). Modeling complex entities with algebraic data types and destructuring them with pattern matching is older than most devs.

Martin Odersky @odersky · Jun 21

Yes, but the connection of OOP and data oriented programming was arguably first made in Scala.

Brian Goetz @BrianGoetz · Jun 21

I don't disagree that Scala has moved the ball forward here (as did other languages, and as Java will too). I am just soooooo tired of people acting like their favorite language invented everything, and injecting that claim into every conversation not about their language.

Brian Goetz
@BrianGoetz

Replying to @BrianGoetz @odersky and @adamwarski

I'll say it louder, in case anyone thinks otherwise: I've read Matching Objects with Patterns more times than I can count, and I am happy to stand on the shoulders of giants, and hopefully to move the art forward another increment.

https://twitter.com/BrianGoetz/status/1539319234915880961

That’s all. I hope you enjoyed that.

@philip_schwarz