The Expression Problem

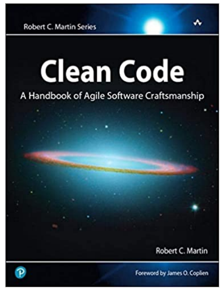
learn about the **expression problem** by looking at

both the **strengths**/**weaknesses** of basic **OOP**/**FP**

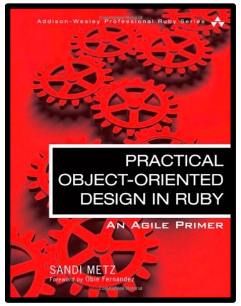
and the role of **different types** of **polymorphism**

Part 1

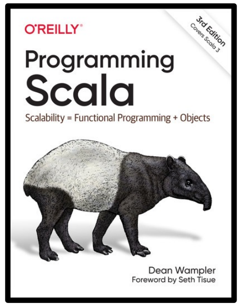
through the work of



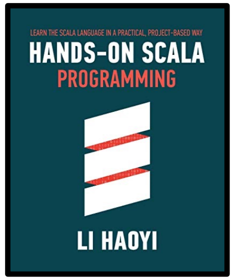
Robert Martin
@unclebobmartin



Sandi Metz
@sandimetz



Dean Wampler
@deanwampler



Li Haoyi
@lihaoyi

Ryan Lemmer

Java Scala Haskell

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

@philip_schwarz

We begin by looking at how **Robert Martin** explains what he calls the **Data/Object anti-symmetry**.

Data/Object Anti-Symmetry

...the difference between objects and data structures.

Objects hide their data behind abstractions and expose functions that operate on that data.

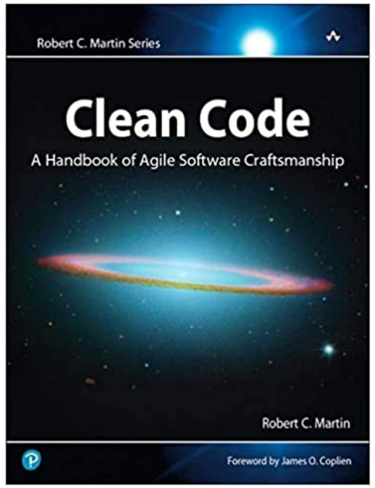
Data structures expose their data and have no meaningful functions.

Go back and read that again. Notice the complementary nature of the two definitions. They are virtual opposites. This difference may seem trivial, but it has far-reaching implications.

Consider, for example, the procedural shape example in Listing 6-5. The Geometry class operates on the three shape classes. The shape classes are simple data structures without any behavior. All the behavior is in the Geometry class.

```java
public class Geometry {
  public final double PI = 3.141592653589793;
  public double area(Object shape) throws NoSuchShapeException {
    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
    }
    else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle)shape;
      return r.height * r.width;
    }
    else if (shape instanceof Circle) {
      Circle c = (Circle)shape;
      return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
```

```java
public class Square {
  public Point topLeft;
  public double side;
}

public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}

public class Circle {
  public Point center;
  public double radius;
}
```



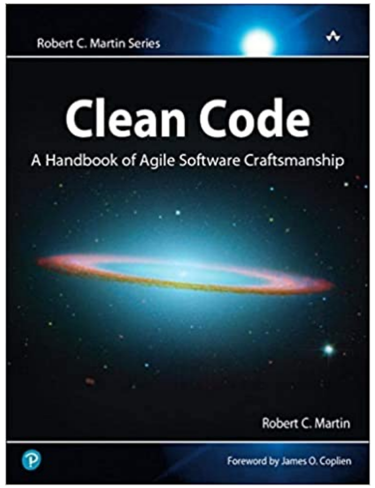
Robert Martin
@unclebobmartin

Java

```java
public class Geometry {

  public final double PI = 3.141592653589793;

  public double area(Object shape) throws NoSuchShapeException {

    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
    }
    else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle)shape;
      return r.height * r.width;
    }
    else if (shape instanceof Circle) {
      Circle c = (Circle)shape;
      return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
```

```java
public class Square {
  public Point topLeft;
  public double side;
}

public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}

public class Circle {
  public Point center;
  public double radius;
}
```

Java



Robert Martin
@unclebobmartin

Object-oriented programmers might wrinkle their noses at this and complain that it is procedural—and they'd be right. But the sneer may not be warranted.

Consider what would happen if a perimeter() function were added to Geometry.

The shape classes would be unaffected! Any other classes that depended upon the shapes would also be unaffected!

On the other hand, if I add a new shape, I must change all the functions in Geometry to deal with it.

Again, read that over. Notice that the two conditions are diametrically opposed.

Here is one way to exercise that code.

By the way, for our current purposes, we can just use the java.awt Point.

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

  var geometry = new Geometry();

  var origin = new Point(0,0);

  var square = new Square();
  square.topLeft = origin;
  square.side = 5;

  var rectangle = new Rectangle();
  rectangle.topLeft = origin;
  rectangle.width = 2;
  rectangle.height = 3;

  var circle = new Circle();
  circle.center = origin;
  circle.radius = 1;

  try {

    if (geometry.area(square) != 25)
      throw new AssertionError("square assertion failed");

    if (geometry.area(rectangle) != 6)
      throw new AssertionError("rectangle assertion failed");

    if (geometry.area(circle) != geometry.PI)
      throw new AssertionError("circle assertion failed");

  } catch (NoSuchShapeException e) {
    e.printStackTrace();
  }
}
```

```java
public class Square {
  public Point topLeft;
  public double side;
}

public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}

public class Circle {
  public Point center;
  public double radius;
}
```

Let’s **modernise** the **procedural program** by using a **sealed interface** and **records**.

As we’ll see later, when we look at excerpts from **Haskell Design Patterns**, this approach in which the **area function** is **dispatched** over the **alternations** (**alternatives**?) of the **Shape type**, is called alternation-based ad-hoc polymorphism.

```java
sealed interface Shape { }
record Square(Point topLeft, double side) implements Shape { }
record Rectangle (Point topLeft, double height, double width) implements Shape { }
record Circle (Point center, double radius) implements Shape { }
```

```java
public class Geometry {

  public final double PI = 3.141592653589793;

  public double area(Object shape) throws NoSuchShapeException {
    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
    }
    else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle)shape;
      return r.height * r.width;
    }
    else if (shape instanceof Circle) {
      Circle c = (Circle)shape;
      return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
}
```

```java
public class Geometry {

  public final double PI = 3.141592653589793;

  public double area(Shape shape) {
    return switch(shape) {
      case Square s -> s.side() * s.side();
      case Rectangle r -> r.height() * r.width();
      case Circle c -> PI * c.radius() * c.radius();
    };
  }

}
```

Java

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

  var geometry = new Geometry();

  var origin = new Point(0,0);

  var square = new Square();
  square.topLeft = origin;
  square.side = 5;

  var rectangle = new Rectangle();
  rectangle.topLeft = origin;
  rectangle.width = 2;
  rectangle.height = 3;

  var circle = new Circle();
  circle.center = origin;
  circle.radius = 1;

  try {

    if (geometry.area(square) != 25)
      throw new AssertionError("square assertion failed");

    if (geometry.area(rectangle) != 6)
      throw new AssertionError("rectangle assertion failed");

    if (geometry.area(circle) != geometry.PI)
      throw new AssertionError("circle assertion failed");

  } catch (NoSuchShapeException e) {
    e.printStackTrace();
  }
}
```

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

  var geometry = new Geometry();

  var origin = new Point(0,0);

  var square = new Square(origin,5);

  var rectangle = new Rectangle(origin,2,3);

  var circle = new Circle(origin,1);

  if (geometry.area(square) != 25)
    throw new AssertionError("square assertion failed");

  if (geometry.area(rectangle) != 6)
    throw new AssertionError("rectangle assertion failed");

  if (geometry.area(circle) != geometry.PI)
    throw new AssertionError("circle assertion failed");
}
```

@philip_schwarz

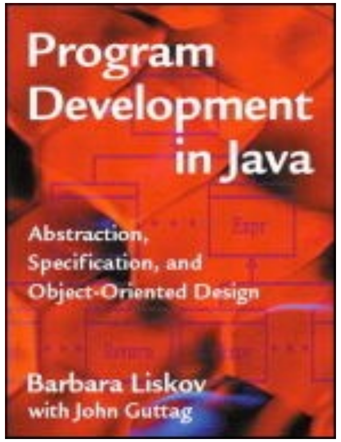
As **Robert Martin** said earlier, **object-oriented programmers might wrinkle their noses at the original procedural program**.

Now that we have switched from using **instanceof** to using **pattern-matching**, **object-oriented programmers might wrinkle their noses at the use of pattern matching**.

Because we now have a **Shape interface** implemented by **Circle**, **Rectangle** and **Square**, an **OO** programmer could object that by **pattern-matching** on the **subtype** of **Shape**, we are violating the **Liskof Substitution Principle** (see the next three slides for a refresher on this principle).

```java
public class Geometry {

  public final double PI = 3.141592653589793;

  public double area(Shape shape) {
    return switch(shape) {
      case Square s -> s.side() * s.side();
      case Rectangle r -> r.height() * r.width();
      case Circle c -> PI * c.radius() * c.radius();
    };
  }

}
```

Java



2000

Let **q(x)** be a **property provable** about **objects x** of type **T**.

Then **q(y)** should be **provable** for **objects y** of type **S** where **S** is a **subtype** of **T**.

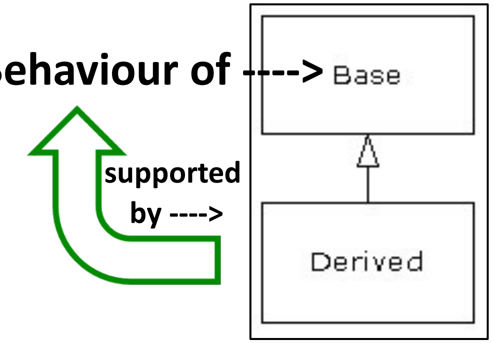
The Liskov Substitution Principle (LSP) - 1988

Barbara Liskov

S
O
LISKOV
I
D

[in a **Type hierarchy**] the **supertype**'s behavior must be supported by the **subtypes**: **subtype objects can be substituted for supertype objects without affecting the behavior of the using code**.

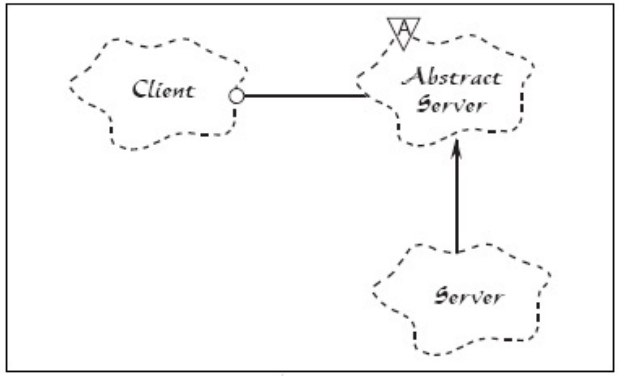
Behaviour of ---->

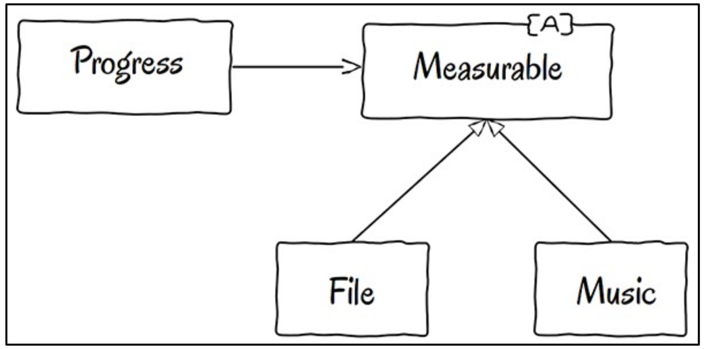


Substitutability

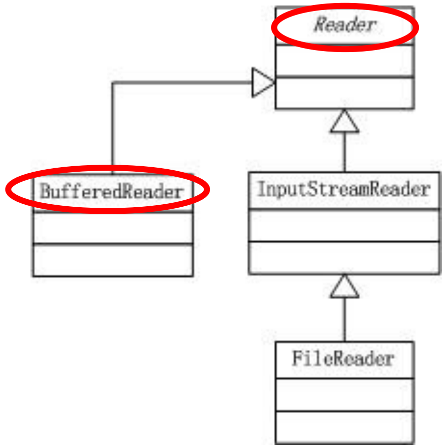
[the LSP] allows **using code** to be **written in terms of the supertype specification**, **yet work correctly when using objects of the subtype**.

Barbara Liskov

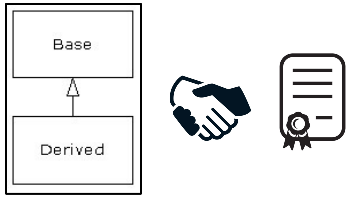




For example, code can be written in terms of the **Reader** type, yet work correctly when using a **BufferedReader**.

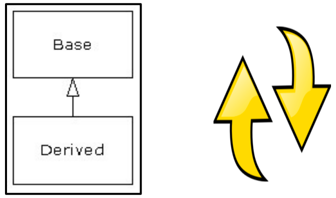



```
private void foo(BufferedReader bufferedReader) throws IOException
{
    …
    bar(bufferedReader);
    …
}

private void bar(Reader reader) throws IOException
{
    …
    System.out.println( reader.read() );
    …
}
```



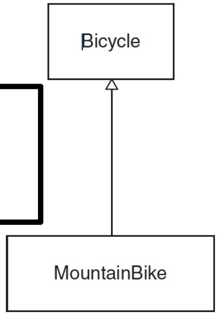
Subclasses agree to a contract

they promise to be substitutable for their superclasses.

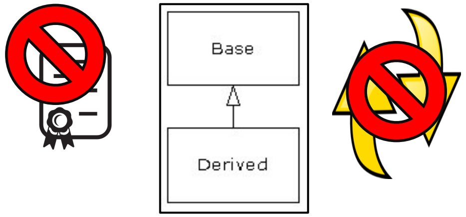


Subclasses are not permitted to do anything that forces others to check their type in order to know how to treat them or what to expect of them.

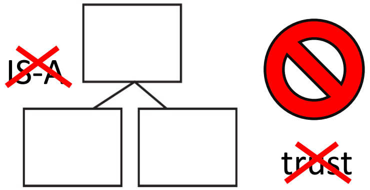
```
if (bicycle instanceof MountainBike)
{
  // special treatment
}
```



Subclasses that **fail to honor their contract** are difficult to use. They're "special"and cannot be freely substituted for their superclasses.

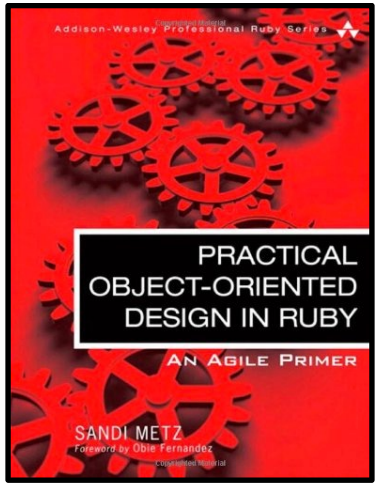


These subclasses are declaring that they are not really a kind-of their superclass and cast doubt on the correctness of the entire hierarchy.



When you honor the contract, you are following the Liskov Substitution Principle, which is named for its creator, Barbara Liskov, and supplies the "L" in the SOLID design principles.

Sandi Metz
@sandimetz



http://www.poodr.com/

The LSP principle's notion of being able to substitute a subtype shape object for a supertype shape object, safe in the knowledge that the behaviour of subtype shape objects does not affect the behaviour of clients of the supertype shape object, is not relevant here because we are not using OO programming: the subtype shape objects have *no* behaviour, they are just anaemic data structures – we are using functional programming-style pattern-matching.

```java
sealed interface Shape { }
record Square(Point topLeft, double side) implements Shape { }
record Rectangle (Point topLeft, double height, double width) implements Shape { }
record Circle (Point center, double radius) implements Shape { }
```

```java
public double area(Shape shape) {
  return switch(shape) {
    case Square s -> s.side() * s.side();
    case Rectangle r -> r.height() * r.width();
    case Circle c -> PI * c.radius() * c.radius();
  };
}
```

Java

Now consider the object-oriented solution in Listing 6-6.

Here the area() method is polymorphic. No Geometry class is necessary.

So if I add a new shape, none of the existing *functions* are affected, but if I add a new function all of the *shapes* must be changed![1]

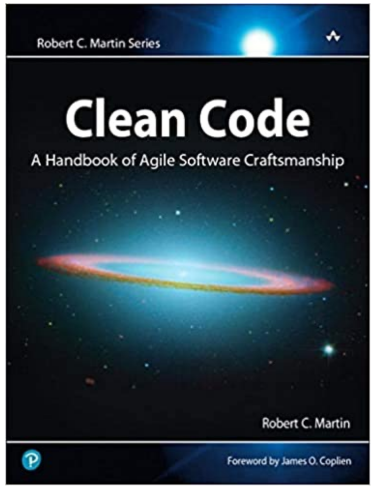
```java
public class Square implements Shape {

  private Point topLeft;
  private double side;

  public double area() {
    return side*side;
  }
}
```

```java
public class Rectangle implements Shape {

  private Point topLeft;
  private double height;
  private double width;

  public double area() {
    return height * width;
  }
}
```

```java
public class Circle implements Shape {

  private Point center;
  private double radius;

  public static final double PI = 3.141592653589793;
  public double area() {
    return PI * radius * radius;
  }
}
```

Java

1. There are ways around this that are well known to experienced object-oriented designers: VISITOR, or dual-dispatch, for example. But these techniques carry costs of their own and generally return the structure to that of a procedural program.

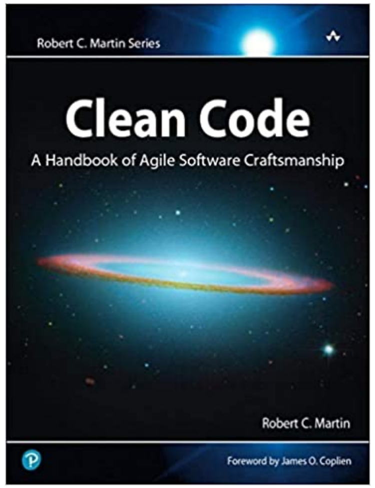


Robert Martin
@unclebobmartin

```java
public class Square implements Shape {

  private Point topLeft;
  private double side;

  public double area() {
    return side*side;
  }
}
```

```java
public class Rectangle implements Shape {

  private Point topLeft;
  private double height;
  private double width;

  public double area() {
    return height * width;
  }
}
```

```java
public class Circle implements Shape {

  private Point center;
  private double radius;

  public static final double PI = 3.141592653589793;
  public double area() {
    return PI * radius * radius;
  }
}
```



Again, we see the complementary nature of these two definitions; they are virtual opposites! This exposes the fundamental dichotomy between objects and data structures:

> *Procedural code (code using data structures) makes it easy to add new functions without changing the existing data structures. OO code, on the other hand, makes it easy to add new classes without changing existing functions.*

The complement is also true:

> *Procedural code makes it hard to add new data structures because all the functions must change. OO code makes it hard to add new functions because all the classes must change.*

So, the things that are hard for OO are easy for procedures, and the things that are hard for procedures are easy for OO!
In any complex system there are going to be times when we want to add new data types rather than new functions. For these cases objects and OO are most appropriate.

On the other hand, there will also be times when we'll want to add new functions as opposed to data types. In that case procedural code and data structures will be more appropriate.

Mature programmers know that the idea that everything is an object *is a myth*. Sometimes you really *do* want simple data structures with procedures operating on them.

Robert Martin
@unclebobmartin

Here is that code again, with some missing bits added, and a **Main** class, to exercise the code.

We said earlier that in the **procedural** code we are using **alternation-based ad-hoc polymorphism**.

In this **OO** code instead, we are using **subtype polymorphism**, in which **subtypes** of **Shape** (implementations of the **Shape** **interface**) **override** (**implement**) methods defined in the **supertype**.

Java

```java
public interface Shape {
  public double area();
}
```

```java
public class Square implements Shape {

  private Point topLeft;
  private double side;

  public Square(Point topLeft, double side){
    this.topLeft = topLeft;
    this.side = side;
  }

  public double area() {
    return side*side;
  }
}
```

```java
public class Circle implements Shape {
  private Point center;
  private double radius;

  public Circle(Point center, double radius){
    this.center = center;
    this.radius = radius;
  }

  public static final double PI = 3.141592653589793;

  public double area() {
    return PI * radius * radius;
  }
}
```

```java
public class Rectangle implements Shape {

  private Point topLeft;
  private double height;
  private double width;

  public Rectangle(Point topLeft, double height, double width){
    this.topLeft = topLeft;
    this.height = height;
    this.width = width;
  }

  public double area() {
    return height * width;
  }
}
```

```java
public class Main {

  public static void main(String[] args) {

    var origin = new Point(0,0);
    var square = new Square(origin, 5);
    var rectangle = new Rectangle(origin, 2, 3);
    var circle = new Circle(origin, 1);

    if (square.area() != 25)
      throw new AssertionError("square assertion failed");
    if (rectangle.area() != 6)
      throw new AssertionError("rectangle assertion failed");
    if (circle.area() != Circle.PI)
      throw new AssertionError("circle assertion failed");
  }

}
```

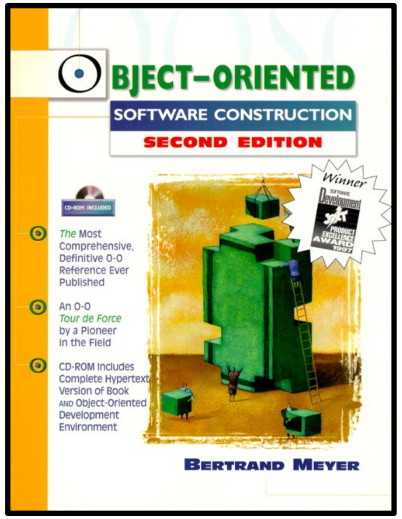
Now let's look at how the **Open Closed Principle** relates to what we have seen so far.

The Open-Closed Principle (OCP)

Bertrand Meyer
@Bertrand_Meyer

Modules should be both open and closed

A module is
- Open if it is still available for extension
- Closed if it is available for use by other modules



1988

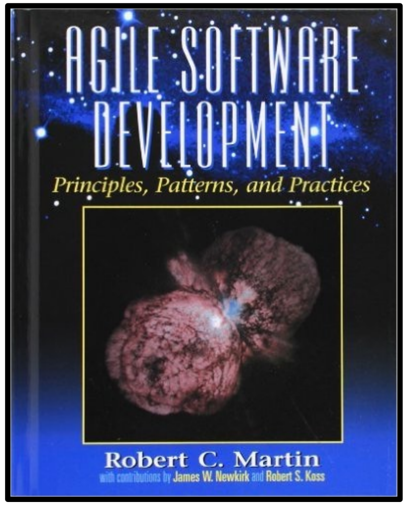
Robert Martin
@unclebobmartin

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

Modules that conform to OCP have two primary attributes:

- They are open for extension. This means that the behavior of the module can be extended. **As the requirements of the application change**, we can extend the module with new behaviors that satisfy those changes. In other words, **we are able to change what the module does.**

- They are closed for modification. **Extending the behavior of a module does not result in changes to the source, or binary, code of the module.** The binary executable version of the module…remains untouched.



2002

Is code using the type of **polymorphism** shown below, **OPEN and CLOSED** with respect to the type of **addition** shown on the right?

		Addition of new	
		Function	Type
Polymorphism	Subtype	OCP✕	OCP✓
	Alternation-based ad-hoc	OCP✓	OCP✕

@philip_schwarz

Closely related to the **data/object anti-symmetry** described by **Robert Martin** in **Clean Code**, is something that **Dean Wampler** writes in **Programming Scala**, on the subjects of **pattern-matching** and **subtype polymorphism**.

A Sample Application

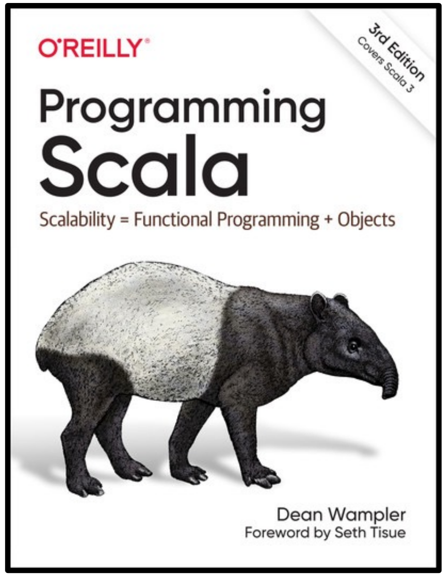
Let's finish this chapter by exploring several more seductive features of **Scala** using a sample application. **We'll use a simplified hierarchy of geometric shapes, which we will send to another object for drawing on a display. Imagine a scenario where a game engine generates scenes. As the shapes in the scene are completed, they are sent to a display subsystem for drawing.**

To begin, we define a Shape class hierarchy:

Dean Wampler
@deanwampler

```scala
case class Point(x: Double = 0.0, y: Double = 0.0)

abstract class Shape():
  /**
   * Draw the shape.
   * @param f is a function to which the shape will pass a
   * string version of itself to be rendered.
   */
  def draw(f: String => Unit): Unit = f(s"draw: $this")

case class Circle(center: Point, radius: Double) extends Shape

case class Rectangle(lowerLeft: Point, height: Double, width: Double) extends Shape

case class Triangle(point1: Point, point2: Point, point3: Point) extends Shape
```

The idea is that callers of **draw** will pass a function that does the actual **drawing** when given a **string representation** of the **shape**. For simplicity, we just use the **string** returned by **toString**, but a structured format like **JSON** would make more sense in a real application.



You could say that draw defines a *protocol* that all shapes have to support, but users can customize. It's up to each **shape** to **serialize** its state to a **string representation** through its **toString** method. The f method is called by **draw**, which constructs the final string using an *interpolated string*.

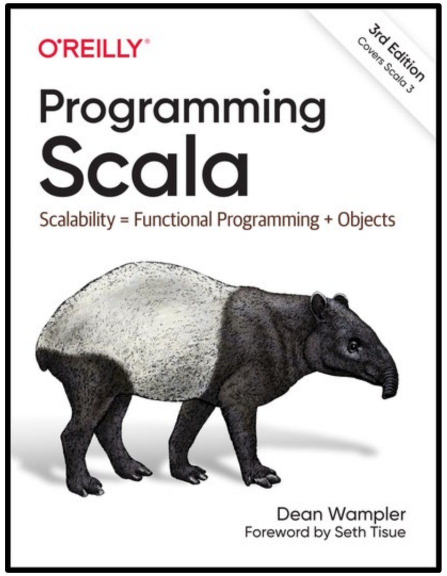
Even though this will be a **single-threaded** application, let's anticipate what we might do in a **concurrent** implementation by defining a set of possible **Messages** that can be exchanged between modules:

```
sealed trait Message
case class Draw(shape: Shape) extends Message
case class Response(message: String) extends Message
case object Exit extends Message
```

Dean Wampler
@deanwampler

The sealed keyword means that we can only define subtypes of Message in the same file. This prevents bugs where users define their own Message subtypes that would break the code we're about to see in the next file! These are all the allowed messages, known in advance.

Recall that Shape was not declared sealed earlier because we intend for people to create their own subtypes of it. There could be an infinite number of Shape subtypes, in principle. So, use sealed hierarchies when all the possible variants are fixed.



If the case clauses don't cover all possible values that can be passed to the match expression, a MatchError is thrown at runtime.

Fortunately, the compiler can detect and warn you that the case clauses are not exhaustive, meaning they don't handle all possible inputs. Note that our sealed hierarchy of messages is crucial here.

If a user could create a new subtype of Message, our match expression would no longer cover all possibilities. Hence, a bug would be introduced in this code!

```
object ProcessMessages:
  def apply(message: Message): Message =
    message match
      case Exit =>
        println(s"ProcessMessage: exiting...")
        Exit
      case Draw(shape) =>
        shape.draw(str => println(s"ProcessMessage: $str"))
        Response(s"ProcessMessage: $shape drawn")
      case Response(unexpected) =>
        val response = Response(s"ERROR: Unexpected Response: $unexpected")
        println(s"ProcessMessage: $response")
        response
```

One of the tenets of OOP is that you should never use if or match statements that match on instance type because inheritance hierarchies evolve.

When a new subtype is introduced without also fixing these statements, they break.

Instead, polymorphic methods should be used.

So, is the pattern-matching code just discussed an antipattern?

Dean Wampler
@deanwampler

PATTERN MATCHING VERSUS SUBTYPE POLYMORPHISM

Pattern matching plays a central role in FP just as subtype polymorphism (i.e., overriding methods in subtypes) plays a central role in OOP.

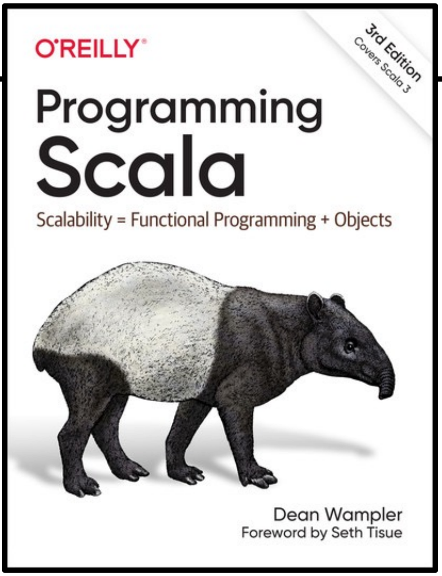
The combination of functional-style pattern matching with polymorphic dispatch, as used here, is a powerful combination that is a benefit of a mixed paradigm language like Scala.

```
object ProcessMessages:
  def apply(message: Message): Message =
    message match
      case Exit =>
        println(s"ProcessMessage: exiting...")
        Exit
      case Draw(shape) =>
        shape.draw(str => println(s"ProcessMessage: $str"))
        Response(s"ProcessMessage: $shape drawn")
      case Response(unexpected) =>
        val response = Response(s"ERROR: Unexpected Response: $unexpected")
        println(s"ProcessMessage: $response")
        response
```

Our match expression only knows about Shape and draw. We don't match on specific subtypes of Shape. This means our code won't break if a user adds a new Shape to the hierarchy.

In contrast, the case clauses match on specific subtypes of Message, but we protected ourselves from unexpected change by making Message a sealed hierarchy. We know by design all the possible Messages exchanged.

Hence, we have combined polymorphic dispatch from OOP with pattern matching, a workhorse of FP. This is one way that Scala elegantly integrates these two programming paradigms!



Closely related to the **data/object anti-symmetry** described by **Robert Martin** and to **Dean Wampler**'s writings on **pattern-matching** and **subtype polymorphism**, is **Li Haoy**'s great explanation, in **Hands-On Scala Programming**, of the different **use cases** for **normal traits** versus **sealed traits**.

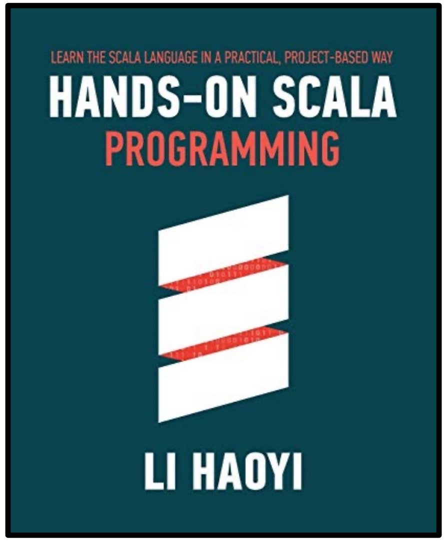
3.4.1 Traits

traits are similar to interfaces in traditional object-oriented languages: a set of methods that multiple classes can inherit. Instances of these classes can then be used interchangeably.

```
trait Point:
  def hypotenuse: Double

class Point2D(x: Double, y: Double) extends Point:
  def hypotenuse = math.sqrt(x * x + y * y)

class Point3D(x: Double, y: Double, z: Double) extends Point:
  def hypotenuse = math.sqrt(x * x + y * y + z * z)

@main def main: Unit =
  val points: Array[Point] = Array(new Point2D(1, 2), new Point3D(4, 5, 6))
  for (p <- points) println(p.hypotenuse)
```

Above, we have defined a **Point trait** with a single method **def hypotenuse: Double**. The **subclasses Point2D** and **Point3D** both have different sets of parameters, but they both implement **def hypothenuse**.

Thus we can put both Point2Ds and Point3Ds into our points: Array[Point] and treat them all uniformly as objects with a def hypotenuse method, regardless of what their actual class is.



Li Haoyi
@lihaoyi

5.1.2 Sealed Traits

traits can also be defined sealed, and only extended by a fixed set of case classes. In the following example, we define a sealed trait Point extended by two case classes, Point2D and Point3D:

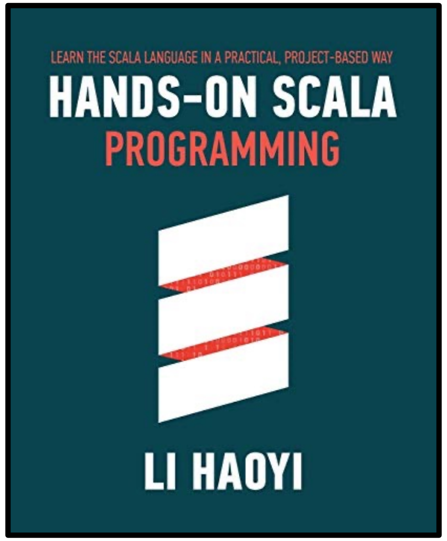
```
sealed trait Point
case class Point2D(x: Double, y: Double) extends Point
case class Point3D(x: Double, y: Double, z: Double) extends Point

def hypotenuse(p: Point) = p match
  case Point2D(x, y) => math.sqrt(x * x + y * y)
  case Point3D(x, y, z) => math.sqrt(x * x + y * y + z * z)

@main def main: Unit =
  val points: Array[Point] = Array(Point2D(1, 2), Point3D(4, 5, 6))
  for (p <- points) println(hypotenuse(p))
```

The core difference between normal traits and sealed traits can be summarized as follows:

- Normal traits are open, so any number of classes can inherit from the trait as long as they provide all the required methods, and instances of those classes can be used interchangeably via the trait's required methods.

- sealed traits are closed: they only allow a fixed set of classes to inherit from them, and all inheriting classes must be defined together with the trait itself in the same file or REPL command …

Because there are only a fixed number of classes inheriting from sealed trait Point, we can use pattern matching in the def hypotenuse function above to define how each kind of Point should be handled.



Li Haoyi
@lihaoyi

5.1.3 Use Cases for Normal v.s. Sealed Traits

Both normal traits and sealed traits are common in Scala applications: normal traits for interfaces which may have any number of subclasses, and sealed traits where the number of subclasses is fixed. Normal traits and sealed traits make different things easy:

- A normal trait hierarchy makes it easy to add additional sub-classes: just define your class and implement the necessary methods. However, it makes it difficult to add new methods: a new method needs to be added to all existing subclasses, of which there may be many.

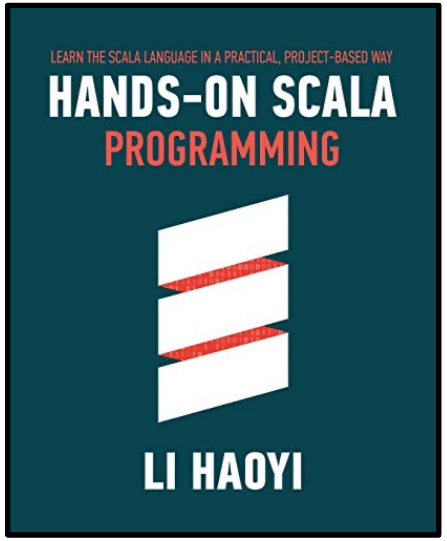
- A sealed trait hierarchy is the opposite: it is easy to add new methods, since a new method can simply pattern match on each sub-class and decide what it wants to do for each. However, adding new sub-classes is difficult, as you need to go to all existing pattern matches and add the case to handle your new sub-class.

In general, sealed traits are good for modelling hierarchies where you expect the number of sub-classes to change very little or not-at-all. A good example of something that can be modeled using sealed trait is JSON:

```
sealed trait Json
case class Null() extends Json
case class Bool(value: Boolean) extends Json
case class Str(value: String) extends Json
case class Num(value: Double) extends Json
case class Arr(value: Seq[Json]) extends Json
case class Dict(value: Map[String, Json]) extends Json
```

- A JSON value can only be JSON null, boolean, number, string, array, or dictionary.
- JSON has not changed in 20 years, so it is unlikely that anyone will need to extend our JSON trait with additional subclasses.
- While the set of sub-classes is fixed, the range of operations we may want to do on a JSON blob is unbounded: parse it, serialize it, pretty-print it, minify it, sanitize it, etc.

Thus it makes sense to model a JSON data structure as a closed sealed trait hierarchy rather than a normal open trait hierarchy.



Li Haoyi
@lihaoyi

Let’s translate our two **Java** programs into **Scala**, starting with the **OO** program.

```java
public interface Shape {
  public double area();
}
```

```scala
trait Shape:
  def area: Double
```

```java
public class Square implements Shape {

  private Point topLeft;
  private double side;

  public Square(Point topLeft, double side){
    this.topLeft = topLeft;
    this.side = side;
  }

  public double area() {
    return side*side;
  }
}
```

```scala
class Square(topLeft: Point, side: Double) extends Shape:
  def area: Double = side * side;
```

```java
public class Circle implements Shape {
  private Point center;
  private double radius;

  public Circle(Point center, double radius){
    this.center = center;
    this.radius = radius;
  }

  public static final double PI = 3.141592653589793;

  public double area() {
    return PI * radius * radius;
  }
}
```

```scala
class Circle(center: Point, radius: Double) extends Shape:
  def area: Double = PI * radius * radius

object Circle:
  val PI: Double = 3.141592653589793
```

```java
public class Rectangle implements Shape {

  private Point topLeft;
  private double height;
  private double width;

  public Rectangle(Point topLeft, double height, double width){
    this.topLeft = topLeft;
    this.height = height;
    this.width = width;
  }

  public double area() {
    return height * width;
  }
}
```

```scala
class Rectangle(topLeft: Point, height: Double, width: Double) extends Shape:
  def area: Double = height * width
```

```java
public class Main {

  public static void main(String[] args) {

    var origin = new Point(0,0);
    var square = new Square(origin, 5);
    var rectangle = new Rectangle(origin, 2, 3);
    var circle = new Circle(origin, 1);

    if (square.area() != 25)
      throw new AssertionError("square assertion failed");
    if (rectangle.area() != 6)
      throw new AssertionError("rectangle assertion failed");
    if (circle.area() != Circle.PI)
      throw new AssertionError("circle assertion failed");
  }

}
```

```scala
@main def main: Unit =

  val origin = Point(0,0)
  val square = Square(origin, 5)
  val rectangle = Rectangle(origin, 2, 3)
  val circle = Circle(origin, 1)

  assert(square.area == 25, "square assertion failed")
  assert(rectangle.area == 6, "rectangle assertion failed")
  assert(circle.area == Circle.PI, "circle assertion failed")
```

And now let's translate the **procedural** program.

```java
sealed interface Shape { }
record Square(Point topLeft, double side) implements Shape { }
record Rectangle (Point topLeft, double height, double width) implements Shape { }
record Circle (Point center, double radius) implements Shape { }
```

```scala
enum Shape:
  case Square(topLeft: Point, side: Double)
  case Rectangle(topLeft: Point, width: Double, height: Double)
  case Circle(center: Point, radius: Double)
```

```java
public class Geometry {
  public final double PI = 3.141592653589793;
  public double area(Shape shape) {
    return switch(shape) {
      case Square s -> s.side() * s.side();
      case Rectangle r -> r.height() * r.width();
      case Circle c -> PI * c.radius() * c.radius();
    };
  }
}
```

```scala
def area(shape: Shape): Double = shape match
  case Square(_,side) => side * side
  case Rectangle(_,width,height) => width * height
  case Circle(_,radius) => math.Pi * radius * radius
```

```java
public class Main {
  public static void main(String[] args) {
    var origin = new Point(0,0);
    var geometry = new Geometry();
    var rectangle = new Rectangle(origin,2,3);
    var circle = new Circle(origin,1);
    var square = new Square(origin,5);

    if (geometry.area(square) != 25)
      throw new AssertionError("square assertion failed");

    if (geometry.area(rectangle) != 6)
      throw new AssertionError("rectangle assertion failed");

    if (geometry.area(circle) != geometry.PI)
      throw new AssertionError("circle assertion failed");
  }
}
```

```scala
@main def main: Unit =

  val origin = Point(0,0)
  val square = Square(origin, 5)
  val rectangle = Rectangle(origin, 2, 3)
  val circle = Circle(origin, 1)

  assert(area(square) == 25, "square assertion failed")
  assert(area(rectangle) == 6 , "rectangle assertion failed")
  assert(area(circle) == math.Pi, "circle assertion failed")
```

@philip_schwarz

Notice how, in translating the **procedural program**, rather than defining the **sealed trait hierarchy literally**

```
sealed trait Shape
case class Square(topLeft: Point, side: Double) extends Shape
case class Rectangle(topLeft: Point, width: Double, height: Double) extends Shape
case class Circle(center: Point, radius: Double) extends Shape
```

we have defined it using an **equivalent** but **more succinct enum**

```
enum Shape:
  case Square(topLeft: Point, side: Double)
  case Rectangle(topLeft: Point, width: Double, height: Double)
  case Circle(center: Point, radius: Double)
```

Next, let's look at **ad-hoc polymorphism** in **Haskell**, both **alternation-based**, and **class-based**.

Ad-hoc polymorphism

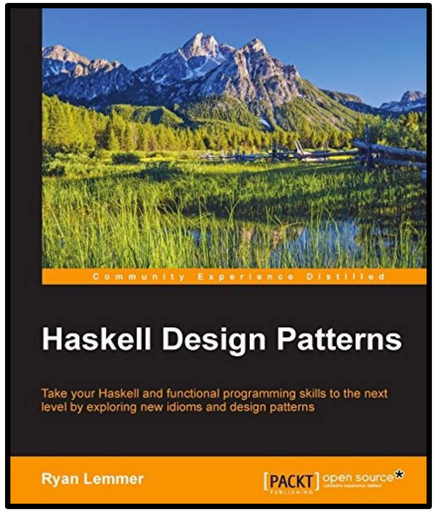
*"Wadler conceived of **type classes** in a conversation with Joe Fasel. Fasel had in mind a different idea, but it was he who had the key insight that **overloading should be reflected in the type of the function**. Wadler misunderstood what Fasel had in mind, and **type classes** were born!"*

-- History of Haskell, Hudak et al.

The canonical example of **ad hoc polymorphism** (also known as **overloading**) is that of the **polymorphic** + operator, defined for all types that implement the **Num typeclass**:

```
class Num a where
  (+) :: a -> a -> a

instance Int Num where
  (+) :: Int → Int → Int
  x + y = intPlus x y

instance Float Num where
  (+) :: Float → Float → Float
  x + y = floatPlus x y
```

In fact, the introduction of **type classes** into **Haskell** was driven by the need to solve the problem of **overloading** numerical operators and equality.



Ryan Lemmer

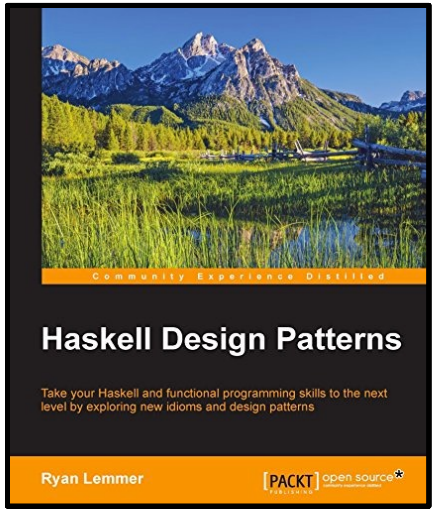
When we call (+) on two numbers, the compiler will **dispatch evaluation** to the **concrete implementation**, based on the **types** of numbers being added:

```
let x_int = 1 + 1         -- dispatch to 'intPlus'
let x_float = 1.0 + 2.5 -- dispatch to 'floatPlus'
let x = 1 + 3.14          -- dispatch to 'floatPlus'
```

In the last line, we are adding what looks like an **int** to a **float**. In many languages, we'd have to resort to explicit **coercion** (of int to float, say) to resolve this type of "mismatch". In **Haskell**, this is resolved by treating the value of 1 as a **type-class polymorphic value**:

```
ghci> :type 1
1 :: Num a => a
ghci>
```

1 is a **generic value**; whether 1 is to be considered an **int** or a **float** value (or a **fractional**, say) depends on the **context** in which it will appear.



Ryan Lemmer

Alternation-based ad-hoc polymorphism

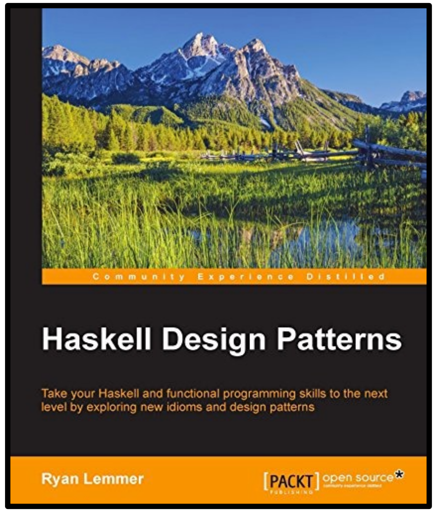
There are two kinds of ad-hoc polymorphism. We've seen the first type already in this chapter:

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

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

The fMaybe function is polymorphically defined over the alternations of Maybe. In order to directly contrast the two kinds of polymorphism, let's carry this idea over into another example:

```haskell
data Shape = Circle Float | Rect Float Float

area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect length width) = length * width
```

The area function is dispatched over the alternations of the Shape type.



Ryan Lemmer

@philip_schwarz

Let’s translate those two **Haskell** examples of alternation-based ad-hoc polymorphism into **Scala**.

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

```
enum Maybe[+A]:
  case Nothing
  case Just(a: A)
```

```
fMaybe :: (a->b) -> Maybe a -> Maybe b
fMaybe f (Just' x) = Just' (f x)
fMaybe f Nothing' = Nothing’
```

```
def fMaybe[A,B](f: A=>B, ma: Maybe[A]): Maybe[B] = ma match
  case Just(a) => Just(f(a))
  case Nothing => Nothing
```

```
data Shape = Circle Float | Rect Float Float
```

```
enum Shape:
  case Circle(radius: Float)
  case Rect(length: Float, width: Float)
```

```
area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect length width) = length * width
```

```
def area(shape: Shape): Double = shape match
  case Circle(radius) => math.Pi * radius * radius
  case Rect(length,width) => length * width
```

Let's see that code again, together with the equivalent Java code.

```haskell
data Shape = Circle Float | Rect Float Float
```

```haskell
area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect length width) = length * width
```

```scala
enum Shape:
  case Circle(radius: Float)
  case Rect(length: Float, width: Float)
```

```scala
def area(shape: Shape): Double = shape match
  case Circle(radius) => math.Pi * radius * radius
  case Rect(length,width) => length * width
```

```java
sealed interface Shape { }
record Circle (double radius) implements Shape { }
record Rect (double height, double width) implements Shape { }
```

```java
public double area(Shape shape) {
    return switch(shape) {
      case Circle c -> PI * c.radius() * c.radius();
      case Rect r -> r.height() * r.width();
    };
  }
```

We can clearly see that adding a new function, e.g. perimeter, would not require us to change any existing code. We would just need to add the code for the new function.

```haskell
perimeter :: Shape -> Float
perimeter (Circle r) = 2 * pi * r
perimeter (Rect length width) = 2 * (length + width)
```

```scala
def perimeter(shape: Shape): Double = shape match
  case Circle(radius) => 2 * math.Pi * radius
  case Rect(length,width) => 2 * (length + width)
```

```java
public double perimeter(Shape shape) {
    return switch(shape) {
      case Circle c -> 2 * PI * c.radius();
      case Rect r -> 2 * (r.height() + r.width());
    };
  }
```

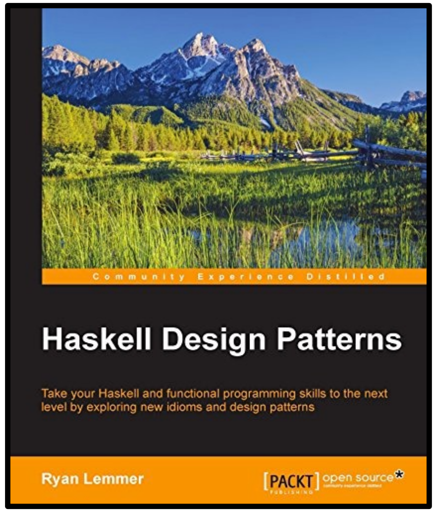
Adding a new Shape, on the other hand, e.g. a Square, would require us to modify all existing functions, e.g. area and perimeter, to get them to handle a Square.

Class-based ad-hoc polymorphism

We could also have achieved a **polymorphic area** function over **shapes** in this way:

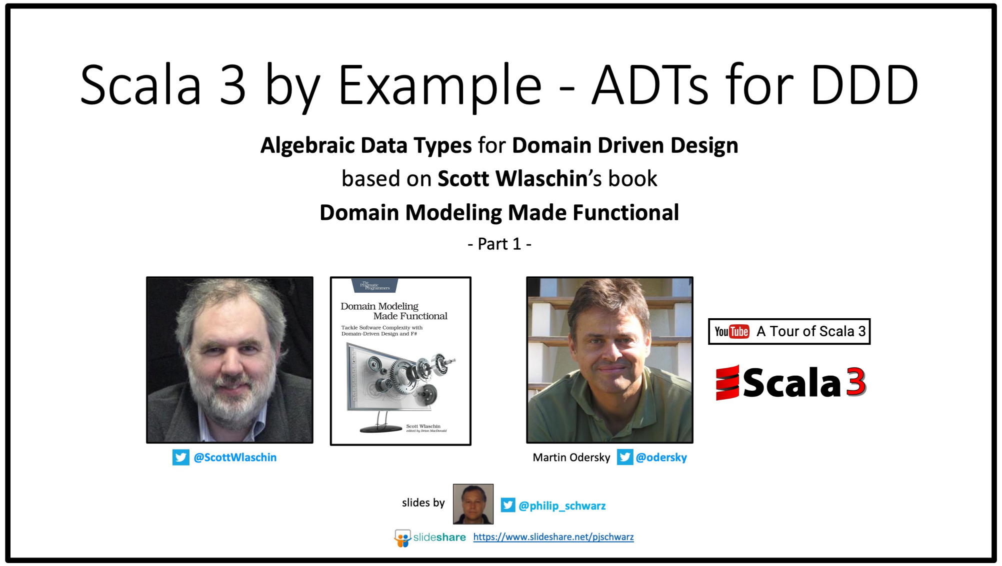
```
data Circle = Circle Float
data Rect = Rect Float Float

class Shape a where
  area :: a -> Float

instance Shape Circle where
  area (Circle r) = pi * r^2

instance Shape Rect where
  area (Rect length' width') = length' * width'
```

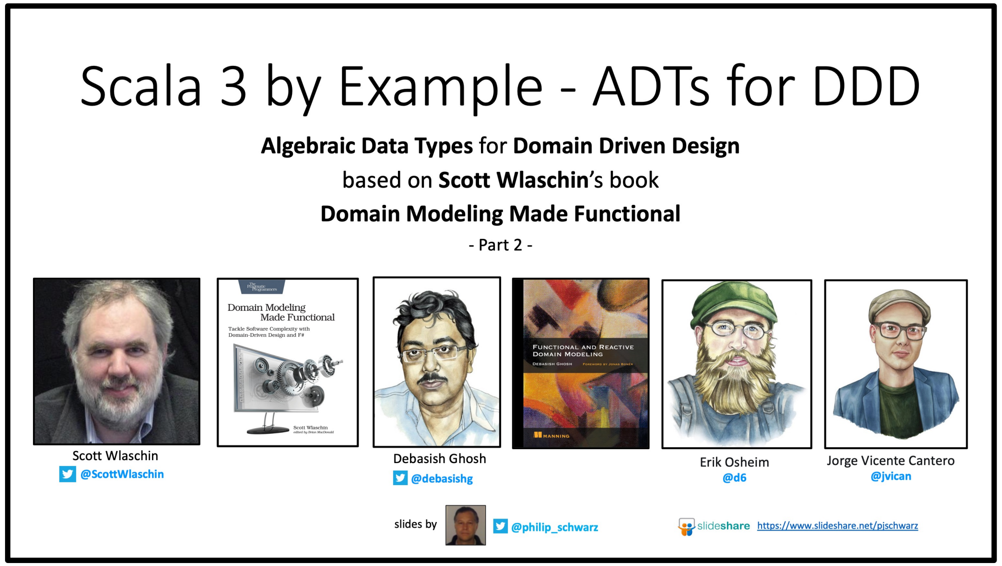
Instead of unifying **shapes** with an **algebraic "sum of types"**, we created two distinct **shape types** and **unified** them with the **Shape type-class**. This time the **area** function exhibits **class-based ad-hoc polymorphism**.

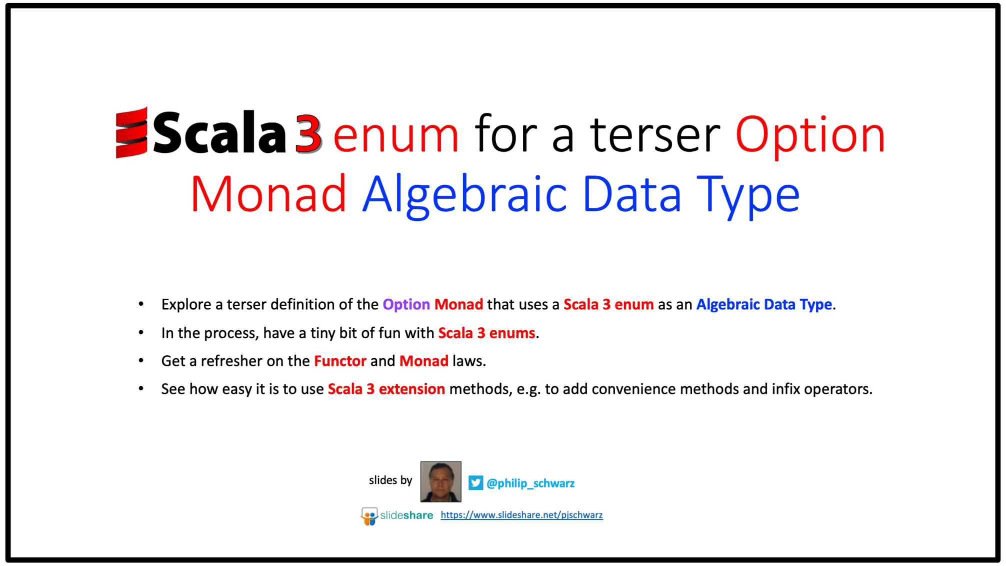


Ryan Lemmer

By the way, if you could do with an introduction to **Algebraic Data Types**, then you might want to take a look at the following:



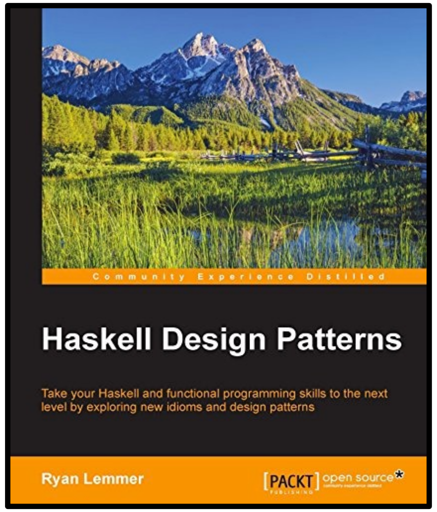




Alternation-based versus class-based

It is tempting to ask "which approach is best?" Instead, let's explore the important ways in which they differ:

	Alternation-based	Class-based
Different coupling between function and type	The function type refers to the algebraic type Shape and then defines special cases for each alternative.	The function type is only aware of the type it is acting on, not the Shape "super type".
Distribution of function definition	The overloaded functions are defined together in one place for all alternations.	Overloaded functions all appear in their respective class implementations. This means a function can be overloaded in very diverse parts of the codebase if need be.
Adding new types	Adding a new alternative to the algebraic type requires changing all existing functions acting directly on the algebraic "super type"	**We can add a new type that implements the type class without changing any code in place (only adding). This is very important since it enables us to extend third-party code.**
Adding new functions	A perimeter function acting on Shape won't be explicitly related to area in any way.	A perimeter function could be explicitly related to area by adding it to the Shape class. This is a powerful way of grouping functions together.
Type expressivity	This approach is useful for expressing simple type hierarchies.	We can have multiple, orthogonal hierarchies, each implementing the type class (For example, we can express multiple-inheritance type relations). This allows for modeling much richer data types.



Ryan Lemmer

Let's translate that **Haskell** example of class-based ad-hoc polymorphism into **Scala**, which also has the concept of a **typeclass**.

```haskell
data Circle = Circle Float
```

```haskell
data Rect = Rect Float Float
```

```haskell
class Shape a where
  area :: a -> Float
```

```haskell
instance Shape Circle where
  area (Circle r) = pi * r^2
```

```haskell
instance Shape Rect where
  area (Rect length' width') = length' * width'
```

```haskell
main = runTestTT
  (TestList [TestCase (assertEqual "test1" pi (area (Circle 1))),
             TestCase (assertEqual "test2" 6  (area (Rect 2 3)))])
```

```scala
case class Circle(radius: Float)
```

```scala
case class Rect(length: Float, width: Float)
```

```scala
trait Shape[A]:
  extension (shape: A)
    def area: Double
```

```scala
given Shape[Circle] with
  extension (c: Circle)
    def area: Double = math.Pi * c.radius * c.radius
```

```scala
given Shape[Rect] with
  extension (r: Rect)
    def area: Double = r.length * r.width
```

```scala
@main def main: Unit =
  assert(Circle(1).area == math.Pi)
  assert(Rect(2,3).area == 6)
```

```haskell
data Circle = Circle Float
```

```haskell
data Rect = Rect Float Float
```

```haskell
class Shape a where
  area :: a -> Float
```

```haskell
instance Shape Circle where
  area (Circle r) = pi * r^2
```

```haskell
instance Shape Rect where
  area (Rect length' width') = length' * width'
```

```scala
case class Circle(radius: Float)
```

```scala
case class Rect(length: Float, width: Float)
```

```scala
trait Shape[A]:
  extension (shape: A)
    def area: Double
```

```scala
given Shape[Circle] with
  extension (c: Circle)
    def area: Double = math.Pi * c.radius * c.radius
```

```scala
given Shape[Rect] with
  extension (r: Rect)
    def area: Double = r.length * r.width
```

We can see clearly that **adding** a **new Shape**, e.g. **Square**, would not require us to **change** any existing code. We would just need to **add** the code for **Square** and a **new Shape typeclass instance** for **Square** providing an **area function** for a **Square**.

```haskell
data Square = Square Float
```

```haskell
instance Shape Square where
  area (Square side) = side * side
```

```scala
case class Square(side: Float)
```

```scala
given Shape[Square] with
  extension (s: Square)
    def area: Double = s.side * s.side
```

Adding a **new function** on the other hand, e.g. **perimeter**, would require us to **modify** the **Shape typeclass** and all existing instances of the **typeclass**, in order to add the **new perimeter function**.

@philip_schwarz

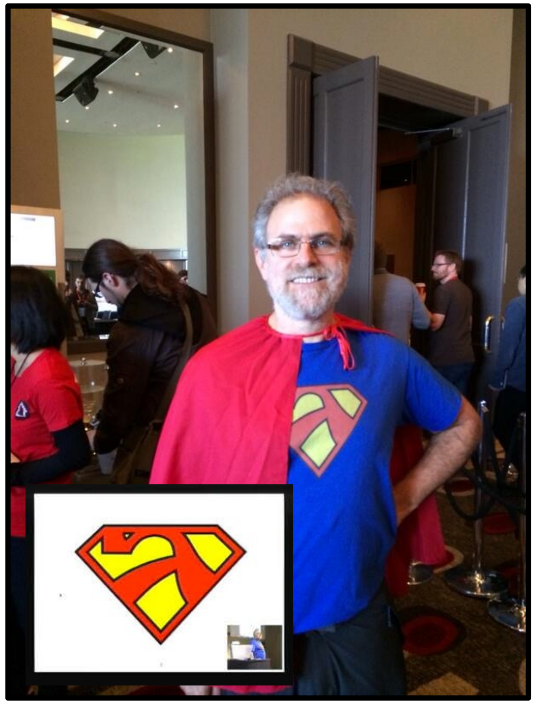
Based purely on the example that we have just seen, it would seem reasonable to add class-based ad-hoc polymorphism to our table in the way shown below.

		Addition of new	
		Function	Type
Polymorphism	Subtype	OCP✗	OCP✓
	Alternation-based ad-hoc	OCP✓	OCP✗
	Class-based ad-hoc	OCP✗	OCP✓

But in actual fact, it turns out that **typeclasses** are more powerful than that. They allow us to solve what is called the **Expression Problem**, i.e. they allow us to write code that is **open and closed** with respect to **both** the **addition** of **new types** and the **addition** of **new functions**.

		Addition of new	
		Function	Type
Polymorphism	Subtype	OCP✕	OCP✓
	Alternation-based ad-hoc	OCP✓	OCP✕
	Class-based ad-hoc	OCP✓	

Here is the definition of the **Expression Problem**.



Computer Scientist **Philip Wadler**

Cc: Philip Wadler <wadler@research.bell-labs.com>
Subject: The Expression Problem
Date: Thu, 12 Nov 1998 14:27:55 -0500
From: Philip Wadler <wadler@research.bell-labs.com>

The **Expression Problem**
Philip Wadler, 12 November 1998

The **Expression Problem** is a new name for an old problem. The goal is to define a **datatype** by **cases**, where one can **add new cases to the datatype** and **new functions over the datatype**, without recompiling existing code, and while retaining static type safety (e.g., no casts). For the concrete example, we take **expressions** as the **data type**, begin with one **case** (**constants**) and one **function** (**evaluators**), then add one more **construct** (**plus**) and one more **function** (**conversion to a string**).

Whether a language can solve the **Expression Problem** is a salient indicator of its capacity for expression. One can think of **cases** as **rows** and **functions** as **columns** in a table. In a **functional language**, the **rows** are fixed (**cases** in a **datatype** declaration) but it is **easy to add new columns** (**functions**). In an **object-oriented language**, the **columns** are fixed (methods in a class declaration) but it is **easy to add new rows** (**subclasses**). We want to make it **easy to add either rows or columns**.

…

https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

@philip_schwarz

In **Part 2** we are going to see how **typeclasses** can be used to solve the **expression problem**.

See you there.