Day 3: ADTs

ADTs - Sum and Product

An algebraic data type is a functional-adjacent, type-safe approach that pairs nicely with pattern matching. An ADT encompasses two approaches:

1. Sum ADT - a `case object` for each constructed member of a particular ADT
2. Product ADT - a `case class` with parameters for extending the domain of your ADT members

ADTs are expressed in Scala with `sealed traits` and `case classes / objects`. Examples:

```
1  // This is a pure sum ADT
2  sealed abstract trait Month
3  case object January extends Month
4  case object February extends Month
5
6  // This is a product ADT
7  sealed case class RGB(red: Int, green: Int, blue: Int)
8
9  // And this is a hybrid approach
10 case class Point(x: Double, y: Double)
11
12 sealed abstract trait Shape
13 case class Circle(centre: Point, radius: Double) extends Shape
14 case class Rectangle(topLeft: Point, height: Double, width: Double) extends Shape
```

Pattern matching against these ADTs is made safe and useful in part because of the `sealed` keyword - this guarantees that the ADT won't be extended to further members outside of the file it's defined in. The compiler can also do some fancy assumptions with this - if you see a `Warning:(19, 5) match may not be exhaustive.`, this is a pattern match against an ADT where not all possible members are covered.

Pattern matching against ADTs allows you to get away with a different kind of compile-time type safety, one that is much looser than strict object-oriented generics and polymorphism. At any point, you can match against an object and see what it is, and then use it as if it's that. In some ways, that feels like a `cast` all over the place, but it's made much safer by having that `sealed` keyword to make it a true ADT.

As usual, a hybrid approach is best.