Day 5: Polymorphism

We define two kinds of polymorphism here:

1. Subtype polymorphism
2. Adhoc polymorphism

Subtype polymorphism

This is your traditional top-down inheritence-based polymorphism. Some member field / method is defined in a super class, and subclasses either give or replace its implementation. This allows you to work with that field / method generically over all its subclasses simultaneously.

```
abstract class Item {
  def pack: String
}

class Fruit extends Item {
  override def pack: String = "I'm a fruit and I'm packed in a bag."
}

class Drink extends Item {
  override def pack: String = "I'm a drink and I'm packed in a bottle."
}

// enables subtype polymorphism like this
// note the type on shoppingBasket is explicitly the parent!
val shoppingBasket: List[Item] = List(new Fruit, new Drink)

shoppingBasket.foreach(i => System.out.println(i.pack))
```

Adhoc polymorphism

This is performed in Scala via the `type class` pattern (`shapeless` is a great library that provides a lot of power to this pattern). In short, instead of run-time top-down polymorphism, we use implicits and compile-time type resolution to provide a more flexible kind of polymorphism (think orthogonal to subtype polymorphism).

In short, we:

1. Provide a behavior that our type class members will implement
2. Provide an entrypoint method to the type class members
3. Provide instances of that behavior for particular types of our type class

```scala
// define the behavior(s) of your type class
trait Adder[T] {
  def sum(a: T, b: T): T
}

// provide entrypoint method
// this makes use of context bounds and `implicitly`, feel free to ask for more info
def sum[T: Adder](a: T, b: T): T =
    implicitly[Adder[T]].sum(a, b)

// provide type class instances for particular types
implicit val int2Adder: Adder[Int] = new Adder[Int] {
  override def sum(a: Int, b: Int): Int = a + b
}

implicit val string2Adder: Adder[String] = new Adder[String] {
  override def sum(a: String, b: String): String = s"$a concatenated with $b"
}
```

This is "adhoc" because we can define as many and as complex type class instances as we went! Again, `shapeless` is great for this.