Day 6: Self Types

Subtype polymorphism is great, but defines purely `is-a` relationships between classes. Sometimes, for separation of concerns, we'd like to define a behavior that `uses-a` / `has-a` different class. This pattern is enabled by `self types`, which look like the following:

```
trait FirstTrait {
  st: SecondTrait =>

  // SecondTrait member fields are available here
}
```

This is different from subtype polymorphism because `FirstTrait` is NOT an instance of `SecondTrait` directly, but it does have access to `SecondTrait` and **cannot** be used without extending `SecondTrait`. Also note that while `FirstTrait` can access all of `SecondTrait`, it **cannot** access the parent fields / methods of `SecondTrait`! This prevents inheritence leaking, a common problem with building re-usable components with only subtype polymorphism.

You can also add multiple traits by doing `_: one with two with ... =>`.

I personally like self types a lot - they're a key component of the `cake pattern` in Scala, and really enable modular component injection. Think of it as injecting a component / behavior at **any** point in an inheritence chain, instead of subtyping and creating really complex / monolithic subtype polymorphism because you have an `is-a` relationship for everything. `has-a` relationships are enabled by self types!

An example of preventing inheritence leaking via self types:

```
trait DB {
  def connect(): Unit = {
    System.out.println("Connected.")
  }

  def dropDatabase(): Unit = {
    System.out.println("Dropping!")
  }

  def close(): Unit = {
    System.out.println("Closed.")
  }
}


```

```scala
trait UserDB {
  this: DB =>

  def createUser(username: String): Unit = {
    connect()
    try {
      System.out.println(s"Creating a user: $username")
    } finally {
      close()
    }
  }

  def getUser(username: String): Unit = {
    connect()
    try {
      System.out.println(s"Getting a user: $username")
    } finally {
      close()
    }
  }
}

trait UserService {
  this: UserDB =>

  // does not compile, but if you did `_: DB =>` it would
  // with direct inheritence, dropDatabase would leak!
  // def bad(): Unit = {
  // dropDatabase()
  //}
}
```