Day 19: The bridge design pattern

The bridge design pattern allows us to design libraries with loosely coupled components we can then evolve independently. In short:

- Identify a component `A` that has multiple implementations
- Ascertain whether a distinct component `B` within `A` also has multiple implementations
- Instead of evolving `A` because of permutations in `B` (leads to class explosion), separate out `B` and inject it into `A` as needed at object creation time.
- This injection is accomplished via constructors in Java, and via self types in Scala. Both accomplish the same thing, but stackable traits are cleaner for adhoc function enhancement at object creation over exploding a constructor.

Code example: say we have a `PasswordConverter` class that is implemented multiple ways, and also uses a `Hasher` that can be implemented multiple ways (SHA256, MD5). Instead of coupling them directly, we use self types to loosely couple them and inject them as needed:

```scala
// component `B` – hasher
trait Hasher {
  def hash(data: String): String
}

trait Sha1Hasher extends Hasher {
  override def hash(data: String): String = s"SHA1-$data"
}

trait Sha256Hasher extends Hasher {
  override def hash(data: String): String = s"SHA256-$data"
}

trait Md5Hasher extends Hasher {
  override def hash(data: String): String = s"MD5-$data"
}

// component `A` which uses `B` – password converter
abstract class PasswordConverter {
  self: Hasher =>

  def convert(password: String): String
}

class SimplePasswordConverter extends PasswordConverter {
  self: Hasher =>

  override def convert(password: String): String = hash(password)
}

class SaltedPasswordConverter(salt: String) extends PasswordConverter {
  self: Hasher =>

  override def convert(password: String): String = hash(s"${salt}:${password}")
}

// top-level injection and management of loosely-coupled components
val p1 = new SimplePasswordConverter with Sha256Hasher
val p2 = new SimplePasswordConverter with Md5Hasher
val p3 = new SaltedPasswordConverter("8jsdf32T^$%") with Sha1Hasher
val p4 = new SaltedPasswordConverter("8jsdf32T^$%") with Sha256Hasher
```

The old-school Java equivalent to this is to use constructors instead of self types. This works fine, but requires you to evolve the constructor of your class `A` to accept as many features as needed. This isn't as nice as just tacking on a bunch of stackable traits!