Day 8: A bit of cake!

Also known as "the bakery of doom", the `cake pattern` is a classic Scala design pattern for injecting composable components using native Scala functionality. As of 2018, I'm not sure if it's still viewed terribly favorably, but it's worth getting an introduction to.

Java manages dependency injection via external libraries (downside: everyone working with those dependencies need to have awareness of the same library), via interfaces (hard to manage lots of them), or via classes being highly parameterized (hard to refactor, easy to make mistakes!).

Let's take a contrived, kinda academic example:

```
1  // basic behaviors
2  trait Hello {
3    def sayhi(msg: String): String
4  }
5
6  trait Wave {
7    def wave(): String
8  }
9
10 trait Greet {
11   def greet(msg: String): String
12 }
13
14 // implementations / components
15 // both a member class and a basic implementation
16 trait HelloComponent {
17   val hello: Hello
18
19   class HelloImpl extends Hello {
20     val iStartWith = "I'mma let you finish, but "
21     override def sayhi(msg: String) = iStartWith + s"$msg"
22   }
23 }
24
25 trait WaveComponent {
26   val wave: Wave
27
28   class WaveImpl extends Wave {
29     override def wave(): String = "I'm waving!"
30   }
31 }
```

```
32
33 trait GreetComponent {
34   this: HelloComponent with WaveComponent =>
35
36   val greet: Greet
37
38   class GreetImpl extends Greet {
39     override def greet(msg: String): String = s"${wave.wave()}\n${hello.sayhi(msg)}\nAnd
   that is my full greeting!"
40   }
41 }
42
43 trait MinimalGreetComponent {
44   this: HelloComponent with WaveComponent =>
45
46   val greet: Greet
47
48   class GreetImpl extends Greet {
49     override def greet(msg: String): String = s"${wave.wave()}\n${hello.sayhi(msg)}"
50   }
51 }
52
53 // bring it all together
54 class GreeterRegistry extends HelloComponent with WaveComponent with MinimalGreetComponent {
55   override val hello = new HelloImpl
56   override val greet = new GreetImpl
57   override val wave = new WaveImpl
58 }
59
60 // use everything brought together!
61 class Greeter extends GreeterRegistry {
62   def greetAndWave(msg: String): String = s"${greet.greet(msg)}\n${wave.wave()}"
63 }
64
65 { new Greeter }.greetAndWave("this seems kinda pointless!")
```

"OH MY GOD WHY" is not a bad thing to be thinking right now. I too still find this pattern a little excessive. So let's try to see what we've gained here, in this "classical cake":

- The basic behaviors (base traits) are well-defined and separated.
- Each behavior's implementation expresses its own dependencies clearly, yet again - separetely.
- The `GreeterRegistry` expresses the `Greeter` class with both a `is-a` and `has-a` relationship to each injected behavior.
- The `GreeterRegistry` could choose any implementation of `hello`, `greet`, and `wave`, as long as they extend the basic behaviors. `GreeterRegistry` is how we express interlinking of dependencies.
- `Greeter` stands as a class with all of its dependencies injected, and now can use all of them at a high level. Think of this as your top-level object, or entrypoint.

Of course, we've also gained a shit-ton of code and complexity. This is a pattern we'll revisit, and as I understand

it better (and how it's aged), I'll try to qualify / expand on this example. For now, key takeaways in my mind:

- The cake pattern is native dependency injection enabled by self types and multiple inheritance.
- All components are injected and resolved at **compile-time**, whereas Java does it at run-time.
- Each component should express some implementation of a behavior, and what it needs in order to actualize that implementation.
- `Registry` classes should bring together dependencies.
- This pattern only really seems worth it if you truly have a ton of different implementations of the same basic idea. Otherwise, simpler subtype / adhoc polymorphism might do just fine.