Day 20: The composite design pattern

This pattern helps to group objects that are commonly used together. In short, provide a common interface and link the implementations such that the user can call the same method and manage the differing objects hierarchically. That's a little opaque, so it's much easier to tether to the classic use case of a tree structure. You could also imagine this for filesystems, AST parsing, etc. It's just barely more than subtype polymorphism - it's that where you expect your subclasses to mingle together in actual use cases, and you want a collection of those subclass objects to play nicely regardless of which one it is, and hide that from the user.

```
1  trait Node {
2    def print(prefix: String): Unit
3  }
4
5  class Leaf(data: String) extends Node {
6    override def print(prefix: String): Unit = println(prefix + data)
7  }
8
9  class Tree extends Node {
10   private val children = scala.collection.mutable.ListBuffer.empty[Node]
11
12   override def print(prefix: String): Unit = {
13     println(prefix + "(")
14     children.foreach(_.print(prefix + prefix))
15     println(prefix + ")")
16   }
17
18   def add(child: Node): Unit = {
19     children += child
20   }
21 }
```

Main thing to note is that both subclasses implement `print`, and the `Tree` implementation can iterate over its children, which can either be a `Tree` or a `Leaf`. Thus, your hierarchy is constructed, and you've done depth-first tree traversal.