Day 32: The memento design pattern

The memento design pattern is all about restoring an object to a previous state. It consists of:

1. **Originator** - an object we want to roll back
2. **Caretaker** - the object that triggers a roll back
3. **Memento** - an object that carries the actual prior state of the originator

Do these all just sound like objects, some with mutable state? Well, Scala is a language that encourages immutability, so in this example we'll lean towards a "Scala-esque" approach that uses a stack for state management. Let's be real, though - a stack is just a fancy way to do mutability, i.e. "I want to use the same reference for deterministically-changing information over time". You better do your threading right, otherwise "deterministically" goes out the window.

Anywho. Let's start with three traits for the above entities:

```
trait Memento[T] {
  protected val state: T
  def getState(): T = state
}

trait Caretaker[T] {
  val states: mutable.Stack[Memento[T]] = mutable.Stack[Memento[T]]()
}

trait Originator[T] {
  def createMemento: Memento[T]
  def restore(memento: Memento[T])
}
```

The `caretaker` manages our mutable stack of state here, whereas the `originator` handle interactions for managing new state.

Our implementation makes the caretaker the point of interaction — an `originator` can deal with a single `memento` instance, whereas a `caretaker` deals with sets of them. It's a pretty standard one-to-many composition situation at the end of the day.

What this means for the originator is this:

```
class TextEditor extends Originator[String] {
  private var builder: StringBuilder = new StringBuilder

```

```
  def append(text: String): Unit = {
    builder.append(text)
  }

  def delete(): Unit = {
    if (builder.nonEmpty) {
      builder.deleteCharAt(builder.length - 1)
    }
  }

  override def createMemento: Memento[String] = new TextEditorMemento(builder.toString)

  override def restore(memento: Memento[String]): Unit = this.builder = new
StringBuilder(memento.getState())

  def text(): String = builder.toString

  private class TextEditorMemento(val state: String) extends Memento[String]
}
```

In words - our actual `memento` is just a given string. The `originator` handles modifications of that string, and can save the current string state at any point. Now, we need a `caretaker` to **receive** the `createMemento` calls, and to pass in the memento to `restore`:

```
class TextEditorManipulator extends Caretaker[String] {
  private val textEditor = new TextEditor

  def save(): Unit = {
    states.push(textEditor.createMemento)
  }

  def undo(): Unit = {
    if (states.nonEmpty) {
      textEditor.restore(states.pop())
    }
  }

  def append(text: String): Unit = {
    save()
    textEditor.append(text)
  }

  def delete(): Unit = {
    save()
    textEditor.delete()
  }

  def readText(): String = textEditor.text()
}
```

Notice the layers of encapsulation here - these objects are all related, but they each have one more layer of responsibility here. "i am one state" -> "i change and save one state" -> "i manage changes of multiple states".

Drawbacks of this pattern are the standard issues with both mutability and stacks (mutability):

- Multi-threaded access might get sticky, especially if your stack objects are mutable!
- Your queue is unbounded - memory?