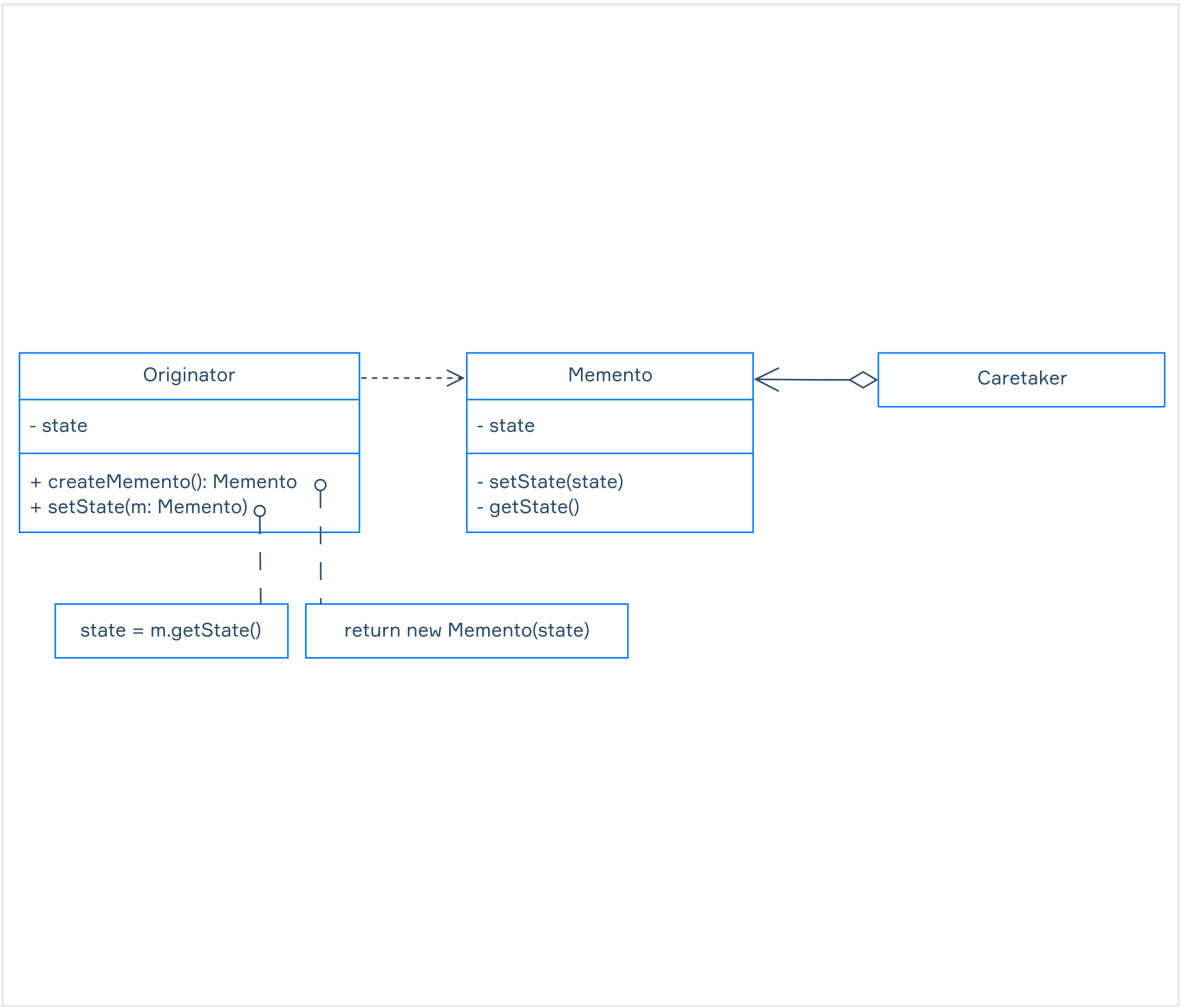
Memento

Sometimes you need to take a snapshot of the state of an object to implement an undo or rollback mechanism. It can be useful if you are working on a text, a graphical editor, or a turn-based game.

Generally, objects encapsulate their internal state to make it inaccessible from the outside, which means you cannot just copy the values of private fields of an object and save them externally. To solve the task and avoid this and some other potential problems, there is a design pattern called **Memento**.

Memento

Memento is one of the classic behavioral patterns and it is intended to facilitate saving and restoring the state of an object without breaking the encapsulation principle. This pattern introduces three objects, the **Originator**, the **Caretaker,** and the **Memento.**



According to this diagram, the Originator is the object whose state, represented by a number of private fields, is to be saved.

The Originator can take snapshots of its state and save them to special objects named Mementos. It can also restore its previous state from a provided Memento.

The Originator can take snapshots of its state and save them to special objects named Mementos.

It can also restore its previous state from a provided Memento.

The Memento is a data object that contains a snapshot of the Originator's state. It does not expose the saved state so that no other object except for the Originator can access it.

In addition, the fields of the Memento may be immutable which prevents the saved state from being accidentally modified.

The Caretaker is an object responsible for calling the respective methods of the Originator to create Mementos and to restore the Originator's state from a saved snapshot.

The Caretaker may also keep a history of the Originator's state changes as a collection of Memento objects.

As you can see, the state of the Originator remains encapsulated and the internal logic of the Originator is simplified since it does not have to decide how and when to save and restore its state.

A drawback of such an approach is high memory consumption in cases when the Originator has to create big Mementos to save its state and the Caretaker has to keep a long history of the Originator's state changes.

Implementation

```
class Editor {
    private String text = "";
    private int textSize = 12;
    private int textColor = 0x000000;

    public void setText(String text) {
        this.text = text;
```

```
    }

    public void setTextSize(int textSize) {
        this.textSize = textSize;
    }

    public void setTextColor(int textColor) {
        this.textColor = textColor;
    }

    @Override
    public String toString() {
        return "Editor{" +
                "text='" + text + '\'' +
                ", textSize=" + textSize +
                ", textColor=" + String.format("0x%06X", textColor) +
                '}';
    }
}
```

With our editor, we can change the text, as well as the text size and color. Also, we can peek at the state of the editor using the overridden toString method. The Editor class will be our Originator.

Next, we need to create a class that will serve as a Memento. Let's call it EditorState.

To preserve encapsulation, we can make it a nested static class inside the Editor class and make its fields and constructor private so that only Editor can access them.

Also, we have to add the required methods to the Editor class to be able to create snapshots and restore its state.

```
class Editor {
    private String text = "";
    private int textSize = 12;
    private int textColor = 0x000000;

    public void setText(String text) { this.text = text; }

    public void setTextSize(int textSize) { this.textSize = textSize; }

    public void setTextColor(int textColor) { this.textColor = textColor; }

    @Override
    public String toString() {...}
```

```
    public EditorState getState() {
        return new EditorState(text, textSize, textColor);
    }

    public void setState(EditorState state) {
        this.text = state.text;
        this.textSize = state.textSize;
        this.textColor = state.textColor;
    }

    static class EditorState {
        private final String text;
        private final int textSize;
        private final int textColor;

        private EditorState(String text, int textSize, int textColor) {
            this.text = text;
            this.textSize = textSize;
            this.textColor = textColor;
        }
    }
}
```

As the next step, let's create a class that will act as the Caretaker:

```
import java.util.ArrayDeque;
import java.util.Deque;

class EditorHistory {
    private final Editor editor;
    private final Deque<Editor.EditorState> history = new ArrayDeque<>();

    EditorHistory(Editor editor) {
        this.editor = editor;
    }

    public void save() {
        history.push(editor.getState());
    }

    public void undo() {
        if (!history.isEmpty()) {
            editor.setState(history.pop());
        }
    }
}
```

It has a reference to our Editor class, a collection of EditorState objects and two methods: save and undo, to manipulate the state of Editor.

Running the code

Now it's time to test our code:

```
class Main {
    public static void main(String[] args) {
        Editor editor = new Editor();
        EditorHistory history = new EditorHistory(editor);

        history.save();
        editor.setText("Hello, world!");

        history.save();
        editor.setTextSize(24);

        history.save();
        editor.setTextColor(0x00FF00);

        System.out.println(editor);
    }
}
```

We created instances of Editor and EditorHistory and made a series of changes to the state of editor, saving a snapshot of it before each change. If we run this code, we will get the following output:

```
Editor{text='Hello, world!', textSize=24, textColor=0x00FF00}
```

Now let's try to undo the changes step by step:

```
history.undo();
System.out.println(editor);

history.undo();
System.out.println(editor);

history.undo();
System.out.println(editor);
```

Here is what we get:

```
Editor{text='Hello, world!', textSize=24, textColor=0x000000}
Editor{text='Hello, world!', textSize=12, textColor=0x000000}
Editor{text='', textSize=12, textColor=0x000000}
```

All the changes are undone one by one!