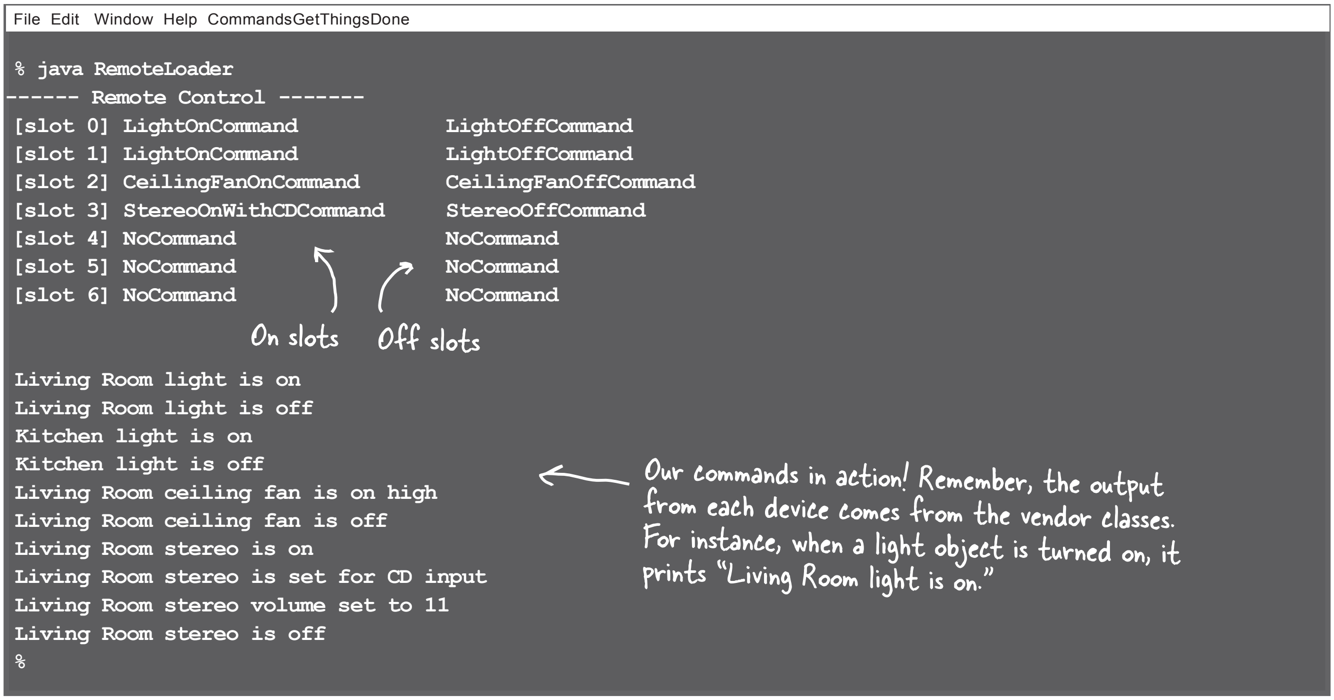
```
        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);
        remoteControl.setCommand(1, kitchenLightOn, kitchenLightOff);
        remoteControl.setCommand(2, ceilingFanOn, ceilingFanOff);
        remoteControl.setCommand(3, stereoOnWithCD, stereoOff);

        System.out.println(remoteControl);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        remoteControl.onButtonWasPushed(1);
        remoteControl.offButtonWasPushed(1);
        remoteControl.onButtonWasPushed(2);
        remoteControl.offButtonWasPushed(2);
        remoteControl.onButtonWasPushed(3);
        remoteControl.offButtonWasPushed(3);
    }
}
```

Now that we've got all our commands, we can load them into the remote slots.

Here's where we use our toString() method to print each remote slot and the command assigned to it. (Note that toString() gets called automatically here, so we don't have to call toString() explicitly.)

All right, we are ready to roll! Now, we step through each slot and push its On and Off buttons.

Now, let's check out the execution of our remote control test...

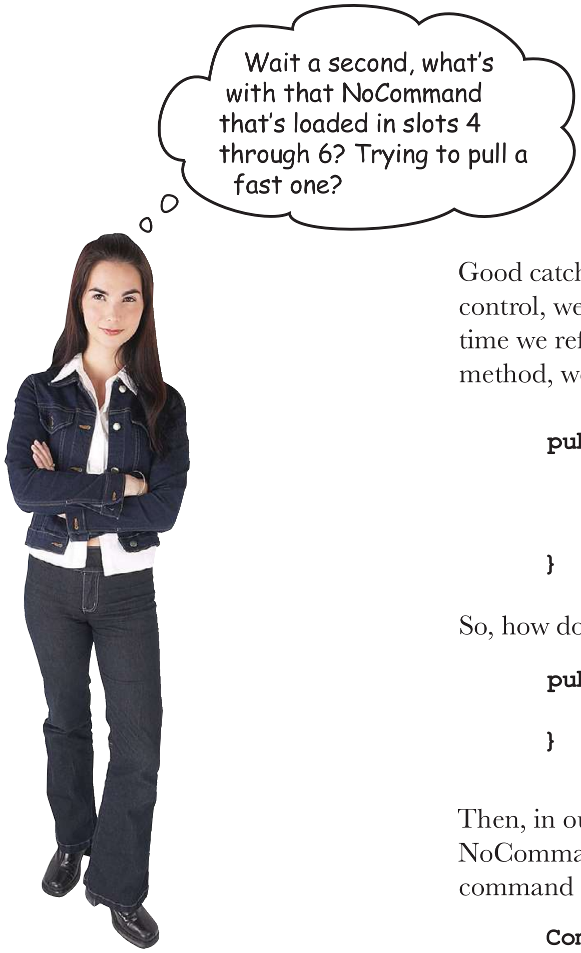


```
% java RemoteLoader
------ Remote Control -------
[slot 0] LightOnCommand             LightOffCommand
[slot 1] LightOnCommand             LightOffCommand
[slot 2] CeilingFanOnCommand        CeilingFanOffCommand
[slot 3] StereoOnWithCDCommand      StereoOffCommand
[slot 4] NoCommand                  NoCommand
[slot 5] NoCommand                  NoCommand
[slot 6] NoCommand                  NoCommand

Living Room light is on
Living Room light is off
Kitchen light is on
Kitchen light is off
Living Room ceiling fan is on high
Living Room ceiling fan is off
Living Room stereo is on
Living Room stereo is set for CD input
Living Room stereo volume set to 11
Living Room stereo is off
%
```

On slots

Off slots

Our commands in action! Remember, the output from each device comes from the vendor classes. For instance, when a light object is turned on, it prints "Living Room light is on."


Wait a second, what's with that NoCommand that's loaded in slots 4 through 6? Trying to pull a fast one?

Good catch. We did sneak a little something in there. In the remote control, we didn't want to check to see if a command was loaded every time we referenced a slot. For instance, in the onButtonWasPushed() method, we would need code like this:

```
public void onButtonWasPushed(int slot) {
    if (onCommands[slot] != null) {
        onCommands[slot].execute();
    }
}
```

So, how do we get around that? Implement a command that does nothing!

```
public class NoCommand implements Command {
    public void execute() { }
}
```

Then, in our RemoteControl constructor, we assign every slot a NoCommand object by default and we know we'll always have some command to call in each slot.

```
Command noCommand = new NoCommand();
for (int i = 0; i < 7; i++) {
    onCommands[i] = noCommand;
    offCommands[i] = noCommand;
}
```

So, in the output of our test run, you're seeing only slots that have been assigned to a command other than the default NoCommand object, which we assigned when we created the RemoteControl constructor.

The NoCommand object is an example of a *null object*. A null object is useful when you don't have a meaningful object to return, and yet you want to remove the responsibility for handling **null** from the client. For instance, in our remote control we didn't have a meaningful object to assign to each slot out of the box, so we provided a NoCommand object that acts as a surrogate and does nothing when its execute() method is called.

You'll find uses for Null Objects in conjunction with many Design Patterns, and sometimes you'll even see "Null Object" listed as a Design Pattern.