state machines in Java

Daniel Jackson

september 10, 2007
topics for today

elements of Java

- classes, objects, fields
- static fields and methods
- constructors
- dynamic dispatch

three patterns for implementing state machines

- machine as class ("singleton"; standard imperative idiom)
- machine as object (standard OO idiom)
- state as object ("state" pattern)
classes and objects
suppose we want a music machine
  • plays regular tones and percussion sounds
  • in response to keypresses
  • can record and playback sequences

strategy
  • model as state machine
  • convert to Java code using a standard pattern
  • we'll see several patterns
operations views

same state machine

' shown in operations view

```plaintext
vars boolean percussionMode;
init !percussionMode
op mouseClicked (k)
post percussionMode' == !percussionMode
op keyPressed (k)
post // play note for key k
```
class MusicMachine {
    static boolean percussionMode = false;
    static void mouseClicked () {
        percussionMode = !percussionMode;
    }
    static void keyPressed (char key) {
    }
}

what's happening here?

• state machine --> class
• state variables --> class or ‘static’ variables
• initialization --> initialization expressions
• operations --> static methods
• postconditions --> assignment statements
machine describing API

how to actually make the sounds?

• use the Java midi API
• for simpler presentation, I've wrapped it

```java
class Midi {
    static void setChannelPercussion () {...}
    static void setChannelRegular () {...}
    static void play (char note) {...}
}
```

putting things together...

class MusicMachine {
    static boolean percussionMode = false;

    static void mouseClicked () {
        if (percussionMode)
            Midi.setChannelRegular ();
        else
            Midi.setChannelPercussion ();
        percussionMode = !percussionMode;
    }

    static void keyPressed (char key) {
        Midi.play(key);
    }
}

using the machine

M.op() calls operation op of machine M
another nesting

how to connect to GUI? like this, but ignore details for now

class MusicApplet extends Applet {

    void init() {
        Midi.init();
        addKeyListener(new KeyAdapter() {
            void keyPressed(KeyEvent e) {
                char key = (char) e.getKeyCode();
                MusicMachine.noteKeyPressed(key);
            }
        });

        addMouseListener(new MouseInputAdapter() {
            void mouseClicked(MouseEvent e) {
                MusicMachine.mouseClicked();
            }
        });
    }
}
class MusicMachine {
    boolean percussionMode = false;

    MusicMachine () {}  // Constructor
    void mouseClicked () {... as before ...}
    void noteKeyPressed (char key) {... as before ...}
}

what's happening here?

• just removed static keyword and added constructor
• state machine --> object
• state variables --> instance variables
• operations --> methods
• this code is a “music machine maker”
instantiating machines

a use of our music machine maker: playing in octaves

```java
class MusicApplet extends Applet {
    void init() {
        Midi.init();
        final MusicMachine m1 = new MusicMachine();
        final MusicMachine m2 = new MusicMachine();
        addKeyListener(new KeyAdapter() {
            void keyPressed(KeyEvent e) {
                char key = (char) e.getKeyCode();
                m1.noteKeyPressed(key);
                m2.noteKeyPressed(octave(key));
            }
        });
    }

    static char octave(char key) {
        return (char) ((int) key + 7);
    }
}
```
class vs. object

machine as class
\- state: static variables
\- operations: static methods
\- `M.op()` calls method `op` of class `M`

machine as object
\- state: instance variables
\- operations: methods
\- `M m = new M()` creates new instance of machine
\- `m.op()` calls method `op` of object `m`
choosing a pattern

which to use?

• machine as class: singleton
• machine as object: standard OO idiom

static elements are bad!

• static vars are global variables
• allow uncontrolled access, so tend to result in more coupling
• static code is inflexible (can’t parallelize, cache, etc)

good uses of singleton

• when there’s really just one of something
• eg, Midi interface
a recording machine

add a new feature

- use R key to start and stop recording
- use P key to playback recording

```plaintext
vars boolean recordMode;
    List[Note] recording;
init !recordMode, recording == <>
op playKeyPressed (k)
post // just play recording
op recordKeyPressed (k)
post recordMode' == !recordMode
    recording' == recordMode ? recording : <>
op noteKeyPressed (k)
post recording' == recording + <k>
```
recording machine in Java

what’s happening here?
- use of ArrayList machine
to be explained later
- import of Java library classes
- type parameterization
- for-loop with iterator

```java
import java.util.*;

class MusicMachine {
    boolean percussionMode = false;
    boolean recordMode = false;
    List<Character> recording = new ArrayList<Character>();

    MusicMachine () {};

    void mouseClicked () {
        if (percussionMode)
            Midi.setChannelRegular ();
        else
            Midi.setChannelPercussion ();
        percussionMode = !percussionMode;
    }

    void noteKeyPressed (char key) {
        Midi.play(key);
        if (recordMode)
            recording.add (key);
    }

    void playKeyPressed () {
        for (char k: recording)
            Midi.play(k);
    }

    void recordKeyPressed () {
        recordMode = !recordMode;
        if (recordMode)
            recording.clear();
    }
}
```
can view array list as a machine

- not the full spec, but useful anyway
- note non-determinism arising from lack of full state

uses

- showing quick overview of methods
- showing when certain exceptions occur
- static analysis of code

---

![Diagram showing transitions between EMPTY and NONEMPTY states with operations like add, clear, remove, and add, remove.](image-url)
inheritance & dynamic dispatch
music machine variants

specializing the music machine

\[\begin{itemize}
\item mouse click switches from percussion to regular and back
\item suppose we want a variant in which it switches octaves instead
\item how to share code between variants?
\end{itemize}\]

inheritance

\[\begin{itemize}
\item base class provides fields and methods
\item subclasses override methods, and add new methods (and fields)
\end{itemize}\]

dynamic dispatch

\[\begin{itemize}
\item which method to run?
\item based on dynamic type of object
\item ie, which constructor made it
\end{itemize}\]
import java.util.List;
import java.util.ArrayList;

class MusicMachineBase {
    boolean recordMode = false;
    List<Character> recording = new ArrayList<Character>();

    MusicMachineBase () {}  
    
    void mouseClicked () {} 
    
    void noteKeyPressed (char key) {
        Midi.play(key);
        if (recordMode)
            recording.add (key);
    }

    void playKeyPressed () {
        if (recordMode) return;
        for (char k: recording)
            noteKeyPressed (k);
    }

    void recordKeyPressed () {
        recordMode = !recordMode;
        if (recordMode)
            recording.clear();
    }
}
class MusicMachinePercussion extends MusicMachineBase {

    boolean percussionMode = false;

    MusicMachinePercussion () {}

    void mouseClicked () {
        if (percussionMode)
            Midi.setChannelRegular ();
        else
            Midi.setChannelPercussion ();
        percussionMode = !percussionMode;
    }
}

this class

• **overrides** mouseClicked (and the constructor)
• **inherits** noteKeyPressed, playKeyPressed, recordKeyPressed
class MusicMachineOctaves extends MusicMachineBase {
    boolean highOctave = false;

    MusicMachineOctaves () {} 

    void mouseClicked () {
        highOctave = !highOctave;
    }

    void noteKeyPressed (char key) {
        super.noteKeyPressed(highOctave ? octave (key) : key);
    }

    static char octave (char key) {
        return (char) ((int) key + 7);
    }
}

note
  · this class overrides mouseClicked and noteKeyPressed
  · noteKeyPressed calls method of superclass
dynamic dispatch

consider this code

MusicMachineBase m = new MusicMachinePercussion ();
m.noteKeyPressed ('2');

how is the method code chosen?

• variable m has declared ('static') type MusicMachineBase
• object bound to m has runtime ('dynamic') type MusicMachinePercussion
• method is 'dynamically dispatched': code from MusicMachinePercussion

MusicMachineBase
  extends
  MusicMachinePercussion
  extends
    MusicMachineOctaves
puzzles

spec of base machine
' why does MusicMachineBase declare an empty mouseClicked?

exposing functionality
' why did MusicMachineBase change playKeyPressed?

```java
// in MusicMachine and MusicMachineBase
void noteKeyPressed (char key) {
    Midi.play(key);
    if (recordMode) recording.add (key);
}

// in MusicMachine
void playKeyPressed () {
    for (char k: recording) Midi.play(k);
}

// in MusicMachineBase
void playKeyPressed () {
    if (recordMode) return;
    for (char k: recording) noteKeyPressed (k);
}
```
fundamental design question: play key in RECORDING mode?
overloading vs. overriding

a common confusion

overriding
  • allows a method to be specialized in a subclass
  • runtime resolves name based on (dynamic) type of receiver object

overloading
  • allows the same name to be used for different but related methods
  • compiler resolves names based on number and (static) types of arguments
  • discussed in next lecture
inheritance

strengths

‣ powerful way to factor out shared code
‣ often more succinct than alternatives
‣ especially useful for frameworks, eg GUIs

weaknesses

‣ much trickier than it seems
‣ subclassing breaks encapsulation
‣ trouble if subclasses aren’t subtypes (more on this later)

our attitude

‣ use inheritance very sparingly
‣ favour composition instead -- see item in Bloch book
interfaces and delegation
interfaces

interface
\• is just a specification: no executable code
\• declares methods with argument and return types
\• claim that class implements interface is checked by compiler

an alternative to subclassing
\• base has no code: just a spec
\• like subclassing: refer to objects by base type, dynamic dispatch
\• unlike subclassing: no inheritance problems

Java interfaces
\• declared like classes
\• but no fields, no constructors, no code in methods
\• class can implement **multiple** interfaces
interface MusicMachine {
    void mouseClicked ()
    void noteKeyPressed (char key)
    void playKeyPressed ()
    void recordKeyPressed ()
}
import java.util.List;
import java.util.ArrayList;

class Recorder {
    boolean recordMode = false;
    List<Character> recording = new ArrayList<Character>();

    void noteKeyPressed (char key) {
        if (recordMode)
            recording.add (key);
    }

    void playKeyPressed (MusicMachine m) {
        if (recordMode) return;
        for (char k: recording)
            m.noteKeyPressed (k);
    }

    void recordKeyPressed () {
        recordMode = !recordMode;
        if (recordMode)
            recording.clear();
    }
}

class MusicMachinePercussion implements* MusicMachine {
    boolean percussionMode = false;
    Recorder recorder = new Recorder();

    void mouseClicked () {
        if (percussionMode)
            Midi.setChannelRegular ();
        else
            Midi.setChannelPercussion ();
        percussionMode = !percussionMode;
    }

    void noteKeyPressed (char key) {
        Midi.play (key);
        recorder.noteKeyPressed(key);
    }

    void playKeyPressed () {
        recorder.playKeyPressed (this);
    }

    void recordKeyPressed () {
        recorder.recordKeyPressed ();
    }
}

*Note: Compiler's check that class implements interface will only be successful if methods of class are declared as public. Access modifiers are explained in the next lecture, and have been omitted in this lecture's slides (but not in the code available in the repository).
class MusicMachineOctaves implements MusicMachine {
    boolean highOctave = false;
    Recorder recorder = new Recorder();

    void mouseClicked () {
        highOctave = !highOctave;
    }

    void noteKeyPressed (char key) {
        Midi.play(highOctave ? octave(key) : key);
        recorder.noteKeyPressed(key);
    }

    void playKeyPressed () {
        recorder.playKeyPressed (this);
    }

    void recordKeyPressed () {
        recorder.recordKeyPressed();
    }

    static char octave (char key) {
        return (char) ((int) key + 7);
    }
}
notes about this example

delegation
- also called composition, forwarding
- share functionality, but each class is self-contained

tricky design question
- how should playback work?
- could provide iterator
- alternative used here: recorder calls back to machine

usage scenario
- no different; client can't see whether inheritance was used

```java
MusicMachine m = new MusicMachinePercussion ();
m.noteKeyPressed ('2');
```
to save code, how about this?

- have MusicMachinePercussion extend Recorder
- method m of MMP calls method m of Recorder with super()

bad idea!

- small saving in code size
- subclass should refine behaviour (ie, spec) of superclass
- if not, client expecting superclass may fail when handed object of subclass
- if add method to Recorder, MMP must offer it too!

earlier version of Java had this bug

- Stack extended Vector
- but a stack is not a vector!
state as object

idea
- suppose we want behaviour to be state dependent
- that is, we have **modes** that determine gross behaviour
- then create an object for each state
- like allowing object to “change class” at runtime

called the “state pattern”
- overkill for our example
- but it's a simple illustration
class MusicMachine {
    MusicMachineState state =
        new PercussionState (new java.util.ArrayList<Character>());

    void mouseClicked () {
        state = state.mouseClicked();
    }

    void noteKeyPressed (char key) {
        state = state.noteKeyPressed(key);
    }

    void playKeyPressed () {
        state = state.playKeyPressed();
    }

    void recordKeyPressed () {
        state = state.recordKeyPressed();
    }
}
state interface

interface
\· defines operations of state machine

interface MusicMachineState {
    MusicMachineState mouseClicked ();
    MusicMachineState noteKeyPressed (char key);
    MusicMachineState playKeyPressed ();
    MusicMachineState recordKeyPressed ();
}
import java.util.List;

class RegularState implements MusicMachineState {
    List<Character> recording;

    RegularState (List<Character> recording) {
        Midi.setChannelRegular ();
        this.recording = recording;
    }
    MusicMachineState mouseClicked() {
        return new PercussionState (recording);
    }
    MusicMachineState noteKeyPressed (char key) {
        Midi.play(key);
        return this;
    }
    MusicMachineState playKeyPressed () {
        for (char k: recording) Midi.play(k);
        return this;
    }
    MusicMachineState recordKeyPressed () {
        return new RecordingState (recording);
    }
}
import java.util.List;

class PercussionState implements MusicMachineState {
    List<Character> recording;

    PercussionState (List<Character> recording) {
        Midi.setChannelPercussion();
        this.recording = recording;
    }

    MusicMachineState mouseClicked() {
        return new RegularState (recording);
    }

    MusicMachineState noteKeyPressed (char key) {
        Midi.play(key);
        return this;
    }

    MusicMachineState playKeyPressed () {
        for (char k: recording)
            Midi.play(k);
        return this;
    }

    MusicMachineState recordKeyPressed () {
        return new RecordingState (recording);
    }
}
import java.util.List;

class RecordingState implements MusicMachineState {
    List<Character> recording;

    RecordingState (List<Character> recording) {
        recording.clear();
        this.recording = recording;
    }

    MusicMachineState mouseClicked() {
        return new RegularState (recording);
    }

    MusicMachineState noteKeyPressed (char key) {
        Midi.play(key);
        recording.add (key);
        return this;
    }

    MusicMachineState playKeyPressed () {
        return this;
    }

    MusicMachineState recordKeyPressed () {
        return new RegularState(recording);
    }
}
notes about state pattern

how many states?

• previous versions: 4 (percussion/regular, recording/playing)
• this version: 3 (percussion, regular, recording)
• pattern is not suited to decomposition of states

when are state variables updated?

• before or after transition to new state?
• in this case, we did both, but have to take extra care

cross-mode state components (eg, recording)

• have to be passed between state objects
• note which classes know about ArrayList usage (see next lecture)
state pattern application

• especially suitable when modes have distinct substates
• example: counting number of keys recorded
• count variable is not even defined in other states

```java
public class RecordingState implements MusicMachineState {
    ...
    int count = 0;

    public MusicMachineState noteKeyPressed (char key) {
        count++;
        ...
    }

    public MusicMachineState recordKeyPressed () {
        System.out.println("Number of keys recorded: " + count);
        return new RegularState(recording);
    }
}
```
review
state machines in Java

key ideas

• subclassing for sharing code
• interfaces for sharing specs
• dynamic dispatch
• delegation vs. inheritance

patterns

• Singleton and State: examples of “Gang of Four” patterns
• stylized idioms used by experienced programmers
• all patterns have advantages and liabilities