Object-Oriented Programming (OOP)
Principles
Applied to a
Sequential Logic Simulator
using Java

Robert Keller
October 2010

**Combinational Logic Elements**

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x ∨ y</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

∨ abbreviates “or”

**Combinational Logic Elements**

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x ∧ y</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

∧ abbreviates “and”

**Combinational Logic Elements**

<table>
<thead>
<tr>
<th>x</th>
<th>¬x</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

¬ abbreviates “not”

**Why “Combinational”**

- These functions are called “combinational” because their output is purely a combination of the current inputs.
- These functions do not have “memory” and thus do not depend on past history.

**Sequential Element**

- An element that does depend on history is the Flip-Flop.
- It always “remembers” its input from the time of the previous “clock tick”.

FF
Flip–Flop Behavior

Assume successive clock ticks are at \( t-1 \) and \( t \).

<table>
<thead>
<tr>
<th>( x(t-1) )</th>
<th>( x(t) )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Example of Sequential Behavior:
- Clock ticks:
  - \( x \)
  - \( y \)

Sequential Logic

- By combining flip–flops with combinational logic, complex sequential behaviors can be achieved.

Sequential Examples

For brevity: false = \( F = 0 \), true = \( T = 1 \)

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence of ( T ), ( F )</td>
<td>( T ) if input was ever ( T ) (( F ) otherwise)</td>
</tr>
<tr>
<td>sequence of 1, 0</td>
<td>1 if three 1’s in a row</td>
</tr>
<tr>
<td>sequence of 1, 0</td>
<td>1 if input had a multiple of 5 1’s</td>
</tr>
<tr>
<td>sequence of 1, 0</td>
<td>1 if input was a multiple of 5 in binary</td>
</tr>
<tr>
<td>sequence of 1, 0</td>
<td>1 if a multiple of 5 in reverse binary</td>
</tr>
</tbody>
</table>

Sequential Logic Implementation Example

- Sequence of \( T \), \( F \) | \( T \) if input was ever \( T \) (\( F \) otherwise) |
- Sequence of 1, 0 | 1 if input was ever 1 (0 otherwise) |

Sequential Logic Implementation Example

- Input Terminal (set from outside)
- Output Terminal (to outside)

Sequential Logic Implementation Example

Initial state:
- 0
- 0
- 0
Sequential Logic Implementation Example

1 input imposed

Clock ticks

change propagates

0 input imposed

Clock ticks
Java-Based Sequential Logic Simulator

“Circuit”

Each node has a label.

in1 or1 FF1 out1

Constructing a Circuit in Java Code

```java
/**
 * Test sequential circuit that remembers whether input was ever true
 */
public void test01l()
{
    sequentialLogic.Circuit circuit = new sequentialLogic.Circuit("test01l");
    circuit.addNode("in01", "inTerminal");
    circuit.addNode("or01", "or");
    circuit.addNode("FF01", "FF");
    circuit.addNode("out01", "outTerminal");
    circuit.connect("in01", "or01");
    circuit.connect("or01", "FF01");
    circuit.connect("FF01", "or01");
    circuit.connect("FF01", "out01");
}
```

Assumptions (for now)

- Each node has a value in \{false, true\}.
- Multiple output connections all see the same value.
- Node functions are symmetric (and, or).
- Multiple input connections represent separate arguments.

Java Class Structure

- We use an “Inheritance Hierarchy” to economize on code and concepts.
- **Class Node** is at the base of the hierarchy:
  - A Node has a name (its label).
  - A Node has a value.
  - A Node can have any number of output connections.
  - The number of input connections is left unspecified in Node.
  - Therefore Node is an abstract class.

Abstract Classes

- An abstract class represents a concept, where there is no intent to construct a member of the class directly.
- Instead, construction is implied by construction of objects lower in the hierarchy.
A Node is a circuit element having a boolean value. It has some number of output Connectors, and possibly some input Connectors. © Robert Keller

abstract public class Node {
    /**
     * the name of this Node.
     */
    private String name;
    /**
     * the current value of this Node
     */
    boolean value = false; // default
    /**
     * the Connectors (0 or more) that connect this Node to
     * other Nodes.
     */
    ArrayList<Connector> outputs;

    protected Node(String name) {
        this.name = name;
        outputs = new ArrayList<Connector>();
    }
}

ArrayList<Connector> is part of the Java library packages.

Connector is a class we define.

ArrayList<Connector> means an ArrayList of Connectors specifically.

Descendants of Node

- OneInputNode
- MultiInputNode

Nodes having only a single input

Nodes allowing 0 or more inputs

- To make use of a library class, we need to import it first into file Node.java:

```java
package sequentialLogic;
import java.util.ArrayList;
abstract public class Node {
    ...
}
```

- Specifying a Descendant

```java
package sequentialLogic;
/**
 * A OneInputNode is a Node that has at most one input. Usually all OneInputNodes have one input, except for
 * InTerminals. It is an Error to attempt to connect more than one Input. © Robert Keller
 */
abstract class OneInputNode extends Node {
    /**
     * The Connector that is input to this Node.
     */
    Connector input = null;
}
```
Still Abstract

- abstract class OneInputNode
  indicates that OneInputNode is also abstract.
- Various specializations of OneInputNode are:
  - Invertor
  - FlipFlop
  - OutTerminal
  - InTerminal
- But only one of these is a Gate. The rest are something else.

The Hierarchy Grows

The hierarchy grows:

```
Node
   \--- OneInputNode
        \---- OneInputGate
               \---- Invertor
               \---- FlipFlop
               \---- OutTerminal
               \---- InTerminal
```

Code for OneInputGate

```
abstract public class Node {
    private String name;
    // the current value of this node
    boolean value = false;        // default
    // the Connectors (0 or more) that connect this node to
    // other Nodes.
    ArrayList<Connector> outputs;
    // Construct a node with no Connectors initially.
    protected Node(String name) {
        this.name = name;
        outputs = new ArrayList<Connector>();
    }
}
```

Constructors in a Hierarchy

- Typically the constructor of a subordinate ("child" or "derived") class will want to call the constructor of a superior ("parent" or "base") class.
- The reason is that each object in the subordinate class is also an object in the superior class.
- The subordinate class refers to its superior’s constructor as super(...) instead of calling it directly.
- The call to super(...) must be the first thing the subordinate does, if it does it at all.

Sub-Class and Super-Class

- Thinking of individual objects as members of a class:
  - The derived or subordinate class is also called the subclass, similar to subset in Math.
  - The base or superior class is also called the superclass, similar to superset in Math.

Constructor Chain

```
abstract public class Node {
    private String name;
    // the current value of this node
    boolean value = false;        // default
    // the Connectors (0 or more) that connect this node to
    // other Nodes.
    ArrayList<Connector> outputs;
    // Construct a node with no Connectors initially.
    protected Node(String name) {
        this.name = name;
        outputs = new ArrayList<Connector>();
    }
    // The Connector that is input to this Node
    protected Connector input = null;
    // Create a OneInputNode with the given name.
    protected OneInputNode(String name) {
        super(name);
        input = new Connector(name);
    }
}
```
Constructor Chain Continues

abstract class OneInputNode extends Node {
    /**
     * The Connector that is input to this Node.
     */
    Connector input = null;

    /**
     * Create a OneInputNode with the given name.
     */
    public OneInputNode(String name) {
        super(name);
    }
}

abstract public class OneInputGate extends OneInputNode {
    /**
     * Construct a OneInputGate with the given name.
     */
    public OneInputGate(String name) {
        super(name);
    }
}

One Object, Many Classes

Node

OneInputNode

MultiInputNode

OneInputGate

An Invertor is a OneInputGate, which is a OneInputNode, which is a Node.

The converse is not true. There are Nodes, that are not MultiInputNodes, etc.

Finally a Non–Abstract Class

public class Invertor extends OneInputGate {
    /**
     * Create an Invertor with the given name.
     */
    public Invertor(String name) {
        super(name);
    }

    /**
     * Update the value of this Invertor by inverting the input.
     */
    public boolean update() {
        value = !input.getValue();
        return super.update();
    }
}

Another Dimension of Classification

• We might want to organize Nodes by Gates vs. other, rather than by number of inputs.

• In our hierarchy there are two branches that have Gates in them, yet there is no common class that has only Gates as subordinates.

• We could try to add a Gate class to the hierarchy. However, then we would have OneInputGate and MultiInputGate having two superiors (called “multiple-inheritance”).

• Multiple inheritance is allowed in some languages (C++, Smalltalk) but not in Java.
**Interfaces**

- The Java **interface** concept is similar to Class as an organizational concept.
- An Interface is like a template defining 0 or more methods by name, but **without** giving implementation code for the methods.
- A class can be declared to implement an interface when it provides those methods, together with working code.

---

**Defining Interfaces in Code**

```java
package sequentialLogic;

interface Gate {
}

abstract public class OneInputNode extends OneInputGate implements Gate {
    /* Construct a OneInputNode with the given name. */
    public OneInputNode(String name) {
        super(name);
    }
}
```

**Using an Interface**

```java
public void showGates() {
    for (Node node : nodes) {
        int gateCounter = 0;
        if (node instanceof Gate) {
            System.out.println("Gate " + ++gateCounter + ": " + node.getName() + " = " + node.getValue());
        }
    }
}
```

---

**Second Dimension, Using Interface**

```
package sequentialLogic;

interface Gate {
}

abstract public class OneInputGate extends OneInputNode implements Gate {
    /* Construct a OneInputGate with the given name. */
    protected OneInputGate(String name) {
        super(name);
    }
}
```

---

**Second Dimension Hierarchy**

(Proposed Gate hierarchy shown in red)

- Gate
  - Node
    - OneInputNode
    - MultiInputNode
    - Invertor
    - FlipFlop
    - OutTerminal
    - InTerminal
    - MultiInputGate
    - OneInputGate
    - AndGate
    - OrGate
    - XorGate
Standard Java Libraries

- The standard Java libraries should be explored as a rich example of class and interface hierarchy.


- Turning “Frames” on is advised

All other classes are subordinate to class Object

String is a direct subclass of Object

An Interface Implemented by String

Privileges Attached to Implementing an Interface

- When a class implements an interface, it automatically accrues privileges:

  - It can be used anywhere a class that implements that interface can be used, e.g. as an argument specifying that interface.
Polymorphism Provided by Interface

```java
public static Character use(CharSequence seq) {
    if (seq.length() > 5) {
        return seq.charAt(5);
    } else {
        return null;
    }
}
```

```java
public static void main(String[] arg) {
    StringBuffer buffer = new StringBuffer();
    String hello = "hello, world";
    buffer.append(hello);
    System.out.println(use(buffer));
    System.out.println(use(hello));
}
```

Polymorphism?

- This is a programming-language term meaning that one type can play the role of several different types.
- In this case, the first type is the Interface, while the other types are classes that implement that interface.