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

Robert Keller
October 2010
Combinational Logic Elements

or-gate

corresponding function ("truth table")

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>x \lor y</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>----------</td>
</tr>
<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>

\lor abbreviates "or"
Combinational Logic Elements

and-gate

and

\[ \text{corresponding function ("truth table")} \]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x \land 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>

\( \land \) abbreviates “and”
Combinational Logic Elements

x \rightarrow \text{not}

invertor

corresponding function (“truth table”)

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

\neg 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: 

$x$: false, false, true, true, false, true, false

$y$: false, true, true, true, false, false, false
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

<table>
<thead>
<tr>
<th>sequence of T, F</th>
<th>T if input was ever T (F otherwise)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence of 1, 0</td>
<td>1 if input was ever 1 (0 otherwise)</td>
</tr>
</tbody>
</table>
Sequential Logic Implementation

Example

Input Terminal (set from outside)

Output Terminal (to outside)
Sequential Logic Implementation

Example

Initial state
Sequential Logic Implementation
Example

1 input imposed
Sequential Logic Implementation

Example

change propagates
Sequential Logic Implementation
Example

Clock ticks
Sequential Logic Implementation Example

0 input imposed

0 → 1 → 1
Sequential Logic Implementation
Example

Clock ticks

0 → 1 → 1
Sequential Logic Implementation

Example

Clock ticks
Java–Based Sequential Logic Simulator

“Circuit”

Each node has a label.

in1  or1  FF1  out1

in  or  FF  out
Constructing a Circuit in Java Code

/**
 * 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");
}

Add nodes “left to right”
Constructing a Circuit in Java Code

/**
  * 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.

```java
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.
   */
```
/**
 * 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>();
}
• **ArrayList** is part of the Java library packages.

• **Connector** is a class we define.

• **ArrayList<Connector>** means an ArrayList of Connectors specifically.
Importing Library Classes

- To make of 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 {
    ... 
```
Descendants of Node

- **OneInputNode**: Nodes having only a single input
- **MultiInputNode**: Nodes allowing 0 or more inputs
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.
 * @author 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

- Invertor
- OneInputGate
- OneInputNode
- FlipFlop
- MultInputNode
- OutTerminal
- InTerminal
Code for OneInputGate
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(...)`. 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.
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>();
    }
}

abstract class OneInputNode extends Node
{
    /**
     * The Connector that is input to this Node.
     */
    
    Connector input = null;
    
    /**
     * Create a OneInputNode with the given name.
     */
    
    protected OneInputNode(String name)
    {
        super(name);
    }

    // Indicates calling constructor of Node, the superior class.
    
    // () indicates calling the constructor of class ArrayList<Connector> with no args.
abstract class OneInputNode extends Node
{
 /**
  * The Connector that is input to this Node.
  */
  
  Connector input = null;

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

abstract public class OneInputGate extends OneInputNode
{
  /**
   * Construct a OneInputGate with the given name.
   */

  public OneInputGate(String name)
  {
    super(name);
  }
}

Indicates calling constructor of Node, the superior class.
Protected

- Protected means that this constructor can be called by a subordinate, but is not generally available (public).
One Object, Many Classes

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

The converse is not true. There are Nodes, that are not MultInputNodes, etc.
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();
    }
}
The Hierarchy Concluded (for now)
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.
Second Dimension Hierarchy
(Proposed Gate hierarchy shown in red)
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.
Interfaces

- Do not confuse Java Interface with the “interface” in, for example, CLI.
- However, an Interface can be used to represent an API.
- A single class can implement any number of Interfaces.
- Interfaces can also be used for grouping classes, apart from any specific methods.
Second Dimension, Using Interface
package sequentialLogic;

interface Gate {
}

abstract public class OneInputGate extends OneInputNode implements Gate {

/**
 * Construct a OneInputGate with the given name.
 */

public OneInputGate(String name) {
    super(name);
}

}

public abstract class MultiInputGate extends MultiInputNode implements Gate {

/**
 * Construct a MultiInputGate with the given name.
 */

protected MultiInputGate(String name) {
    super(name);
}

}
Using an Interface

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

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

- Using a web browser, start at, e.g.
  

- Turning “Frames” on is advised
All other classes are subordinate to class Object.
String is a direct subclass of Object

java.lang
Class String

declares java.lang.Object

All Implemented Interfaces:
Serializable, CharSequence, Comparable<String>

public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

    String str = "abc";
An Interface Implemented by String

java.lang

Interface CharSequence

All Known Subinterfaces:
   Name

All Known Implementing Classes:
   CharBuffer, Segment, String, StringBuffer, StringBuilder

public interface CharSequence

A CharSequence is a readable sequence of char values. This interface provides uniform, read-only access to many different kinds of char sequences. A char value represents a character in the Basic Multilingual Plane (BMP) or a surrogate. Refer to Unicode Character Representation for details.

This interface does not refine the general contracts of the equals and hashCode methods. The result of comparing two objects that implement CharSequence is therefore, in general, undefined. Each object may be implemented by a different class, and there is no guarantee that each class will be capable of testing its instances for equality with those of the other. It is therefore inappropriate to use arbitrary CharSequence instances as elements in a set or as keys in a map.

Since:
   1.4

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>(CharSequence)</td>
</tr>
<tr>
<td>String</td>
</tr>
</tbody>
</table>

These **must** be provided by any class implementing CharSequence.
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;
    }
}

global 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) );
}
```

Doable because both `StringBuffer` and `String` implement `CharSequence`. 
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.