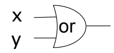
Object-Oriented Programming (OOP) Principles Applied to a Sequential Logic Simulator using Java

Robert Keller
October 2010

Combinational Logic Elements



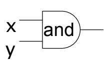
or-gate

corresponding function ("truth table")

x	y	x ∨ y
false	false	false
false	true	true
true	false	true
true	true	true

∨ abbreviates "or"

Combinational Logic Elements



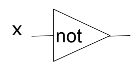
and-gate

corresponding function ("truth table")

x	y	x ∧ y
false	false	false
false	true	false
true	false	false
true	true	true

∧ abbreviates "and"

Combinational Logic Elements



invertor

corresponding function ("truth table")

x	¬x
false	true
true	false

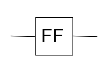
¬ abbreviates "not"

Why "Combinational"

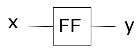
- These functions are called "combinational" because their output is purely a combination of the current inputs.
- These functions do not have "memory" and thus do not depend on past **history**.

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".



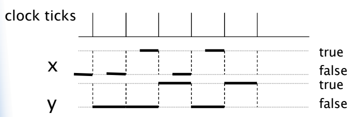
Flip-Flop Behavior



Assume successive clock ticks are at t−1 and t.

x(t−1)	x(t') $t-1 \le t' < t$	y
false	false	false
false	true	false
true	false	true
true	true	true

Example of Sequential Behavior:



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

Input	Output
sequence of T, F	T if input was ever T (F otherwise)
sequence of 1, 0	1 if three 1's in a row
sequence of 1, 0	1 if input had a multiple of 5 1's
sequence of 1, 0	1 if input was a multiple of 5 ***in binary***
sequence of 1, 0	1 if a multiple of 5 ***in reverse binary***

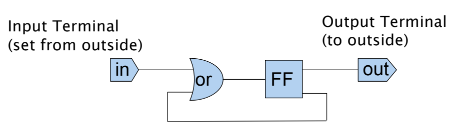
Sequential Logic Implementation Example

sequence of T, F	T if input was ever T (F otherwise)

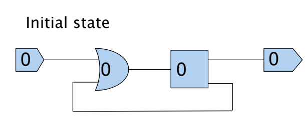
=

sequence of 1, 0	1 if input was ever 1 (0 otherwise)

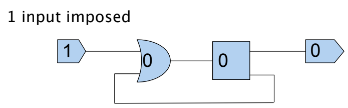
Sequential Logic Implementation Example



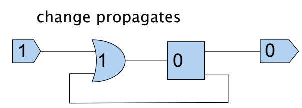
Sequential Logic Implementation Example



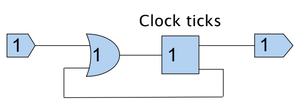
Sequential Logic Implementation Example



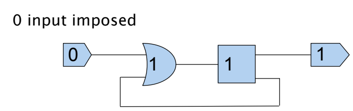
Sequential Logic Implementation Example



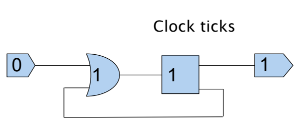
Sequential Logic Implementation Example



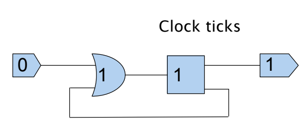
Sequential Logic Implementation Example



Sequential Logic Implementation Example



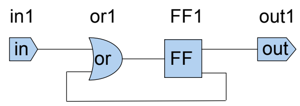
Sequential Logic Implementation Example



Java-Based Sequential Logic Simulator

"Circuit"

Each node has a label.



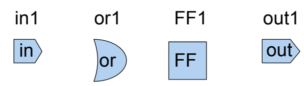
Constructing a Circuit in Java Code

```
/**
 * 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");
```

Package name

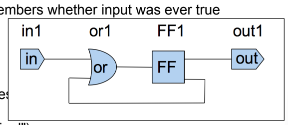
Class name

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("tes

  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.

Start of Node

```
/**
 * A Node is a circuit element having a boolean value.
 * It has some number of output Connectors,
 * and possibly some input Connectors.
 * @author 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.
   */
```

More of Node

```
/**
  * 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<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

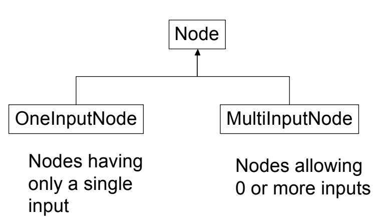
- To make of use of a library class, we need to *import* it first into file Node.java:

Package we're defining containing class Node

Package where ArrayList resides.

```
package sequentialLogic;

import java.util.ArrayList;


abstract public class Node
{
. . .
```

Descendants of Node



Specifying a Descendant

```
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;
```

Indicates inheritance from Node

Still Abstract

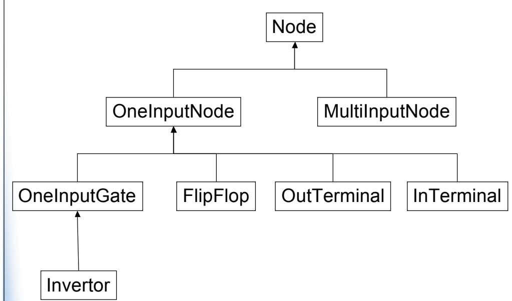
- `abstract class OneInputNode`

  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



Code for OneInputGate

Constructors in a Hierarchy

- Typically the constructor of a subordinate ("child" or "derived") class will want to the 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.

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>();
  }
```

```
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.

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.
  */

 protected OneInputNode(String name)
 {
  super(name);
 }
```

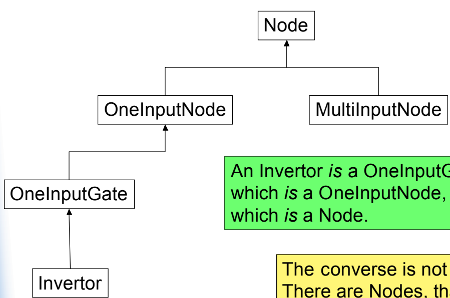
```
abstract public class OneInputGate extends OneInputNode
{
 /**
  * Construct a OneInputGate with the given name.
  */

 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.

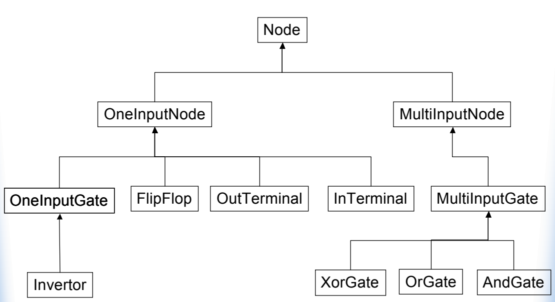
The converse is not true. There are Nodes, that are not MultiInputNodes, etc.

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();
 }

}
```

Indicates calling the update() method of superior

The Hierarchy Concluded (for now)

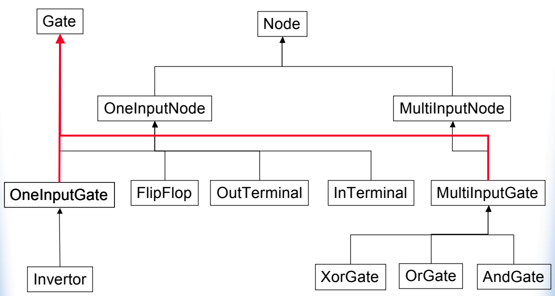


Another Dimension of Classification

- 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)



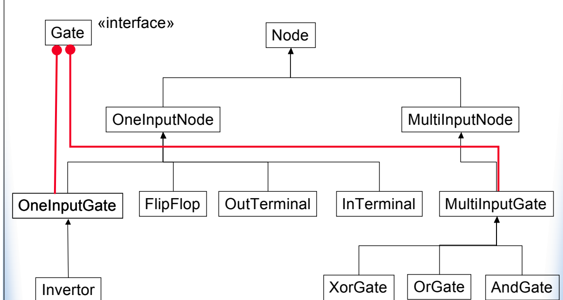
Interfaces

- 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



Defining Interfaces in Code

```
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 MultiInpuGate 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());
    }
 }
 }
```

instanceof [one word] can be used for to detect whether the object in question is a member of the class or interface.

Standard Java Libraries

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

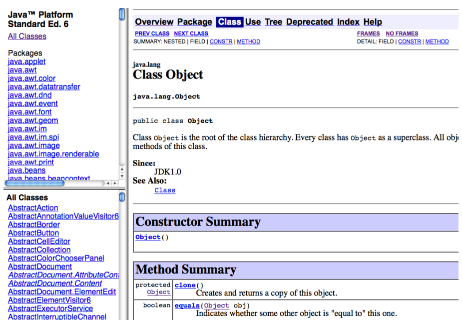
Standard Java Libraries

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

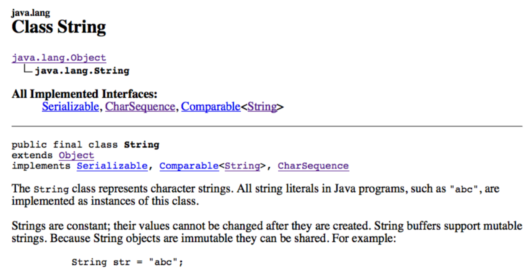
http://download.oracle.com/javase/6/docs/api/index.html?java/lang/Object.html

- Turning “Frames” on is advised

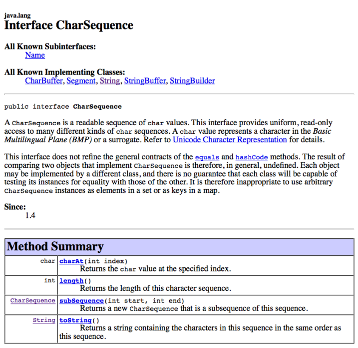
All other classes are subordinate to class Object



String is a direct subclass of Object



An Interface Implemented by String



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

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

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) );
}
```

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.