Object-Oriented Programming CSE-703029

Faculty of Computer Science

Phenikaa University

Lecture 6: Interfaces and Inner Classes

Today’s Topics

- Interfaces: the ultimate in [completely] abstract classes
- Simulating multiple inheritance
- “Stupid interface tricks”
- Inner classes: named, anonymous, and static
- The “callback” idea
- Control frameworks

Interfaces

- An interface is a “pure” abstract class.
- The intent of an interface is to specify a set of methods that a concrete class will honor.
- Class inherits interface is to implement interface.
- New class can implement several interfaces.
- Interfaces have no data members (except **static final**) or no method bodies.

Some Java Interfaces

EventListener

- Methods: none

ActionListener extends EventListener

- Methods: ActionPerformed()

Adjustable (java.awt.Adjustable)

- Methods: get/setMaximum/Minimum(), getValue(), etc.

CharacterIterator

- Methods: first(), last(), next()

Technical details

Use the **interface** keyword.

An **interface** usually sits in its own .java file.

Use **public** or nothing (making it friendly).

- **private** and **protected** aren't allowed. (*no inheritance*)

To implement, use a class with a keyword of **implements**

Interface methods are always **public**.

```
interface Animal{

          public void animalSound();

}
```

```
class Dog implements Animal {
      public void animalSound(){
      System.out.println("Dog is barking");}
}
```

Node as an Interface

```
public interface Node {
    void setChild(int position, Node n);
    double eval();
    String toString();
}
```

- Note there is no constructor given.
- The **abstract** keyword isn’t used for the methods.
- The methods are automatically **public**.

Abstract class

- ❑ **Special type of Class, can not instantiate**

  Animal obj X new Animal();

  Animal is abstract class

- ❑ **Type of Objects**
- ❑ **At least One pure virtual function (function without implementation)**

  public void animalSound();

Binop Can Still be Abstract

```
public abstract class Binop implements Node {
    protected Node lChild, rChild;
    public Binop() {}
    public Binop(Node l, Node r) {
            lChild = l; rChild = r;
    }
//Abstract class can have abstract and concrete methods
//concrete method
    public void setChild(int position, Node n) {
        if (position == 1)
            lChild = n;
        else
            rChild = n;
    }
}
```

	Abstract Class	**Interface**
1	Type of Objects	Set of behaviours
2	Access modifiers: Public, Protected, and Private	Public
3	A Class can inherit from **One** Abstract Class	A Class can implement **multiple** interfaces
4	Have Abstract and Concrete Methods	Only Abstract Methods
5	Can implement Interface	Can NOT implement Abstract Class
6	Keywords: Extends	Keywords: implements
7	Extend only One abstract class and implement multiple interfaces	Can implement multiple interfaces

Make **Binop** an Interface?

- **Binop** currently has the implementation of **setChild()**.
- We would have to place copies of this code in all the classes that implemented **Binop**.
- This is probably unwise ("inelegant and error-prone").
- Note that **Node** is still a *type*, so we use it just as before.

Java’s **Comparable** Interface

```
public class Student implements Comparable {
    public Student(String name, float gpa) {
        this.name = name;
        this.gpa = gpa;
    }
    public Student() {}
    //Object is the ancestor of every class
    public int compareTo(Object o) {
            //cast the correct type (Student)o
        if ( ((Student)o).gpa < gpa )     return 1;
        else if ( ((Student)o).gpa > gpa )    return -1;
        else   return 0;
      }
```

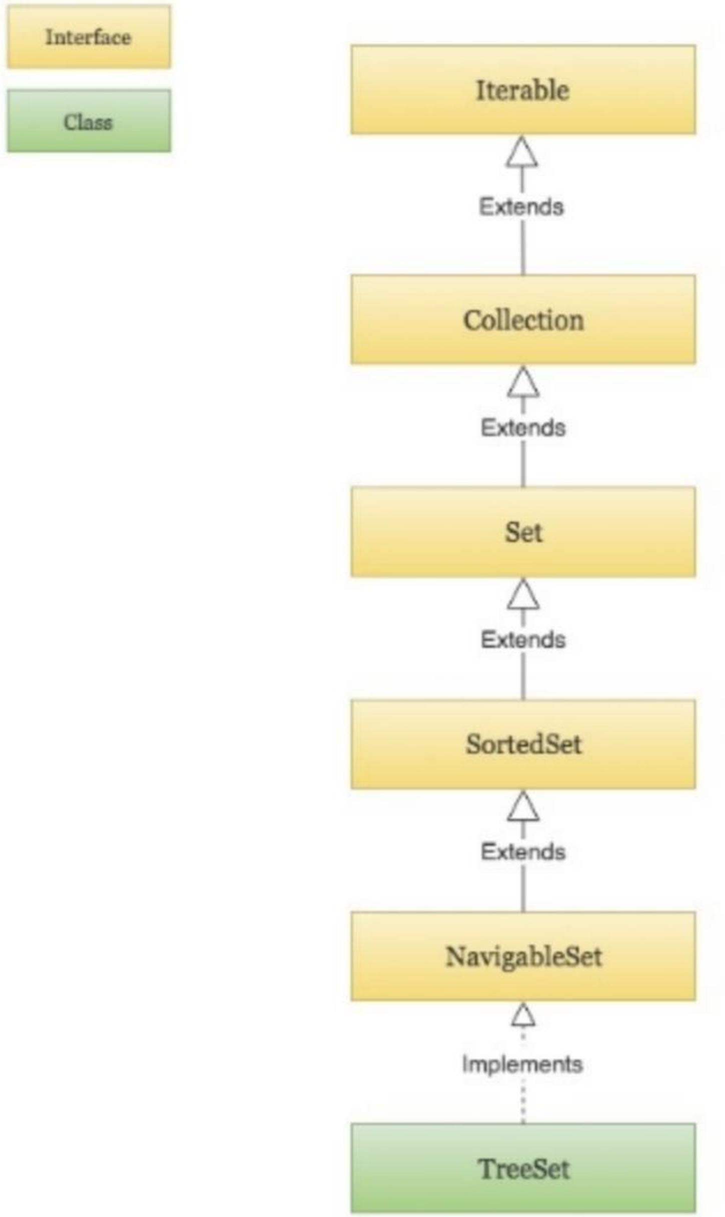
Class Student (cont.)

```
  // Can this equals() be improved and be consistent??
  public boolean equals(Object o) {
  //cast the correct type (Student)o
   if (gpa == ((Student) o).gpa)  return true;
   else return false;
     }
  public String getName() { return name;}
  public float getGpa() { return gpa;}
  private String name;
  private float gpa;
}
```

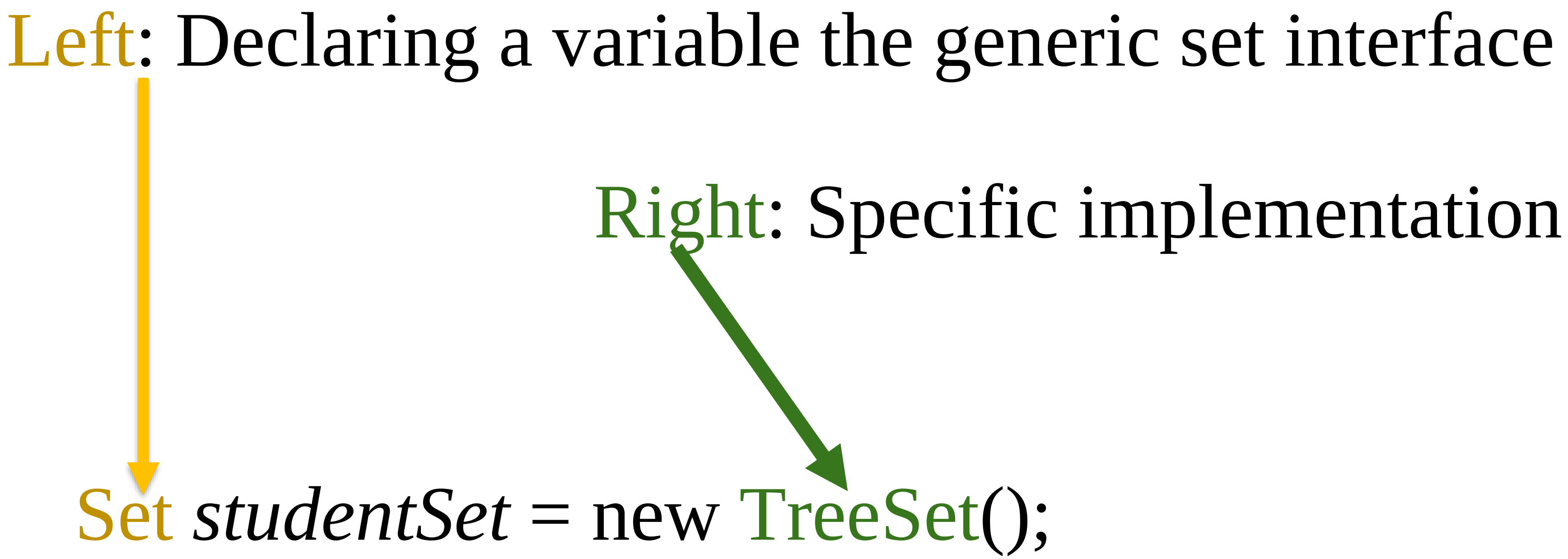
Using Class Student

```
public class TestStudent {
    public static void main(String[] args) {
            Student s1 = new Student("Fred", 3.0F);
            Student s2 = new Student("Sam", 3.5F);
            Student s3 = new Student("Steve", 2.1F);
            if (s3.compareTo((Object)s2) < 0)
               System.out.println(s3.getName() + " has a lower gpa than " + s2.getName());
            //Set is an interface in Java ; TreeSet is its implementation
        Set studentSet = new TreeSet();
            //Method of set add(element)
        studentSet.add(s1); studentSet.add(s2); studentSet.add(s3);
       //Iterator interface access elements of Map, List, Set
        Iterator i = studentSet.iterator();
       //method of Iterator hasNext()
        while(i.hasNext())
           System.out.println( ((Student)i.next()).getName());
    }
}
```

Java TreeSet Class Hierarchy



Interface and its implementation: Style of writing code in Java



Left: Declaring a variable the generic set interface

Right: Specific implementation

Set *studentSet* = new TreeSet();

Comparable and TreeSet

- The **Comparable** interface specifies a single method, **compareTo(Object)**.
- It should be "consistent with equals()", so I defined **equals()** accordingly. But was my **equals()** enough? No, see the notes for the last lecture!
- Since **Student** implements **Comparable**, a "container class" like **TreeSet** knows how to use them (it stores them in order).

Other Interface Issues

- You can inherit from an interface to create a new interface.
- A single class can implement several interfaces simultaneously (Java’s version of multiple inheritance).
- This is OK, since interfaces have no data or method code that could conflict.
- You must watch out for method name clashes, though.

Example of a Name Collision

```
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }
class C2 implements I1, I2 {
    public void f() {}
    public int f(int i) { return 1; } //overloaded
}
class C3 extends C implements I2 {
    public int f(int i) { return 1; } //overloaded
}
class C4 extends C implements I3 {
    // identical, no problem
    public int f() { return 1; }
}
public class Collision extends C implements I1 { }
```

f() in C cannot implement f() in I1; attempting to use incompatible return type

Inner Classes

- It’s possible (and sometimes encouraged!) to define one class within another.
- This provides another way to group classes that work closely together.
- Inner classes can be “shielded” so that they are unknown to the outside world.
- Often inner classes are used to hide a class-specific implementation of an external interface.

A Primitive Iterator

```
public interface Selector {
    boolean end();
    Object current();
    void next();
}
```

- This provides a way to access elements in “container classes.”
- If everyone uses the same interface, new container class types are interchangeable.

A Primitive Container Class

```
public class Sequence {
   private Object[] objects; //array of Object
    private int next = 0;
    public Sequence(int size) { objects = new Object[size]; }
     public void add(Object x) { //x dont have length
                    if (next < objects.length) { objects[next] = x;  next++} }
           //inner class SSelector of Sequence
    private class SSelector implements Selector {
     //Selector is an interface
                    int i = 0;
                    public boolean end() {return i == objects.length  }
                    public Object current() { return objects[i]; }
                    public void next() {if (i < objects.length) i++; }}
                    public Selector getSelector() {  return new SSelector();}
     }
}
```

Testing the Sequence Class

```
public class TestSequence {
    public static void main(String[] args) {
        Sequence s = new Sequence(10);
        for (int i = 0; i < 10; i++)
            s.add(Integer.toString(i));
            //selector is interface
        Selector sl = s.getSelector();
        while(!sl.end()) {
            System.out.println(sl.current());
            sl.next();
        }
    }
}
```

Inner Class

```
//Outerclass
public class OuterClassTest {
    int var1 ;
   public class InnerClassTest {
   public void InnerClassTest(){System.out.println(“var1”+this.var1)}
   }
}
//Main
    public static void main(String[] args) {
        OuterClassTest oct= new OuterClassTest();
        OuterClassTest.InnerClassTest ict = oct.new InnerClassTest();
        System.out.println(ict);
        }
}
```

Sequence and Selector

- This is quite similar to Java’s use of **Iterator**
- The inner class can
  - access the outer class (e.g., get at the array),
  - implement a public interface, and
  - remain private, and specialized.
- We might write a tree-based container, using the same Selector interface.
- It would be easy for clients to switch between the two, if necessary for performance reasons.

Sequence and Selector (cont.)

- Sequence is pretty primitive, eh?
- How would you improve it to
  - be more type-specific?
  - handle overflow more gracefully?
  - access specific elements directly?

More on Inner Classes

- Look again at **SSelector**, for example the line **return objects[i];** in the method **current()**.
- How does the **SSelector** object know which **Sequence** object it “belongs to”?
- It holds a reference to the object that created it. The compiler takes care of these details.
- You can get this reference by saying **outerClass.this**
- You can’t create an inner class object unless you have an outer class object to start with, unless…

Static Inner Classes

- You don’t need an object of the outer class type to create an object of a static inner class type.
- You can’t access an outer class object with a static inner class type (there is no **this**).
- A bottle of Rolling Rock to the first person providing a convincing example of where you’d want to do this!

Anonymous Inner Classes

- If we never need to invoke the name of an inner class type, we can make the inner class *anonymous*.
- This is a common idiom in Java, but somewhat confusing.
- It is much used, for example, in GUI programming with Swing **listener** classes.

Anonymous **Selector** Subclass

```
public class Sequence {
    :
    public Selector getSelector() {
        return new Selector() {
            int i = 0;
            public boolean end() {
            return i == objects.length;
            }
            public Object current() {
            return objects[i];
            }
            public void next() {
            if (i < objects.length) i++;
            }
        };
    }
}
```

This replaces the explicit definition of **SSelector**, whose name we never used.

Selector is an interface, but this idea works with concrete classes too.

SSelector is innerClass

Skip the name to

Anonymous innerClass

Random Facts on Inner Classes

- **.class** files are created, using the convention **OuterClassName$InnerClassName.class**
- Anonymous inner classes are in files like **OuterClassName$1**
- You can nest inner classes indefinitely, but things quickly become confusing.
- You can have multiple inner classes implementing *different* interfaces.

Simple Callback Mechanism

```
interface Incrementable {
    void increment();
}

public class Callee implements Incrementable {
    private int i = 0;
    public void increment() {
        i++;
        System.out.println("Callee increments i to " + i);
    }
}
```

Simple Callback (cont.)

```
public class Caller {
    private Incrementable callbackReference;
    Caller(Incrementable cbr) {
        callbackReference = cbr;
    }
    void go() {
        callbackReference.increment();
    }
}
public class TestCallback {
    public static void main(String[] args) {
        Callee callee = new Callee();
        Caller caller = new Caller(callee);
        caller.go();
    }
}
```

Callback :
Incrementatble is implemented in Callee.
Caller call "Incrementable" back
Caller call Callee

Event-Driven Systems

- Normally we write code with a specific sequence of message passing in mind.
- Sometimes, though, the "outside world" has to determine the sequence of events:
  - responding to server hits
  - responding to external sensors
  - responding to UI activities

Event-Driven Systems (cont.)

- Eckel’s simulation of an event-driven system illustrates this.
- There is an **Event** interface with //Eckel’s built
  - a time to “fire”
  - a constructor
  - a **ready()** method to see if the Event should “fire”
  - an **action()** method performing the event’s responsibility
  - a **description()** method

The **Event** Class

```
public abstract class Event {
   private long eventTime;   // the time for this event to “fire”
   public Event(long evtTime) {
      eventTime = evtTime;
   }
   public boolean ready() {   // is it time to “fire” yet?
      return System.currentTimeMillis() >= eventTime;
   }
   abstract public void action();      // what to do when I “fire”
   abstract public String description();
}
```

The **EventSet** Class

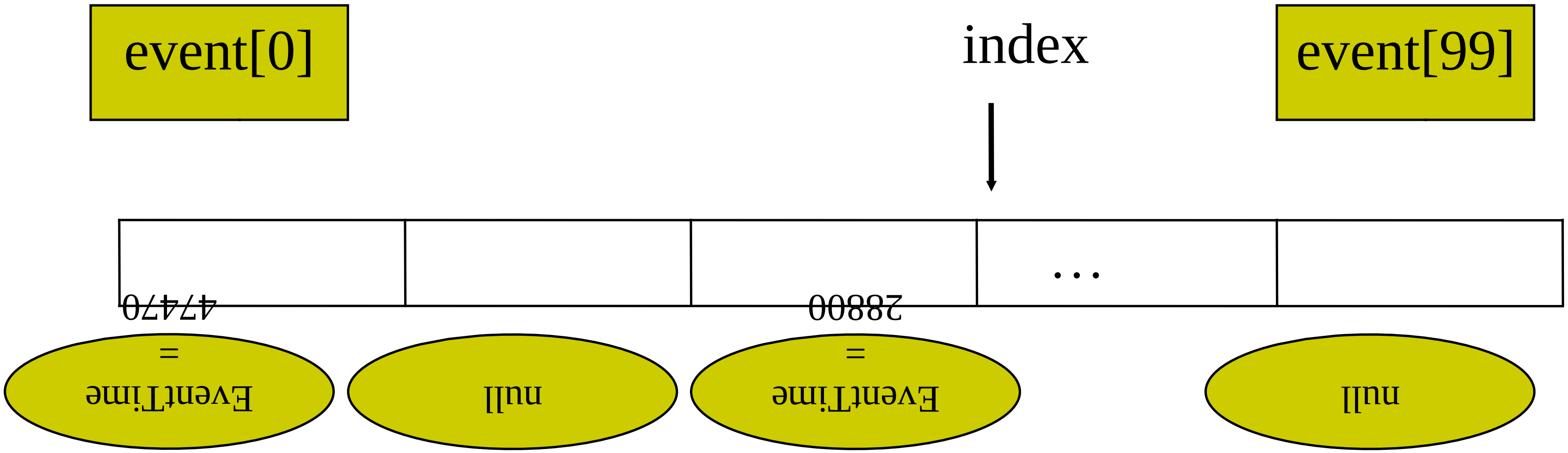
- This holds an array of objects (pending **Events)**, named **events[]**.

  Event **events[]** = new Event[100]
- **events[]** elements refer to either an actual pending **Event**, or to **null**.
- The **getNext()** method looks for a non-null **Event** in **events[]**, and returns it if there is one. It returns null if there are no more events to process.
- This is quite realistic in a GUI-based program.

The Class **EventSet**

```
public class EventSet {
  private Event[] events = new Event[100];
  private int index = 0;         // index for inserting
  private int next = 0;          // next place to look
  public void add(Event e) {
    if (index >= events.length)
      return;                       // Event e isn’t added!
Bad!
      events[index++] = e;
  }
  public void removeCurrent() {
    events[next] = null;         // mark for garbage collection
  }
```

EventSet Structure



EventSet's getNext() Method

```
// return the next pending Event, or null if no more Events
public Event getNext() {
   boolean looped = false;
   int start = next;
   do {                  // look for a non-null
      next = (next + 1) % events.length;
      if (start == next)
         looped = true;
      if ((next == (start + 1) % events.length) && looped)
         return null;
   } while(events[next] == null);
   return events[next];
}
```

Now The Controller Class

```
public class Controller {
   private EventSet es = new EventSet();
   public void addEvent(Event c) { es.add(c); }
   public void run() {
      Event e;
      while ((e = es.getNext()) != null) {
         if (e.ready()) {
            e.action();
            System.out.println(e.description());
            es.removeCurrent();
         }
      }
   }
}
```

Some Event Subclasses

```
private class StartStudying extends Event {
    public StartStudying(long eventTime) {
        super(eventTime);
    }
    public void action() {
            studying = true;
    }
    public String description() {
        return "Can't you see I'm studying Java?";
    }
}
```

Some Event Subclasses

```
 private class StopStudying extends Event {
    public StopStudying(long eventTime) {
        super(eventTime);
    }
    public void action() {
              studying = false;
    }
    public String description() {
        return "I'm sick of studying Java!";
    }
}
```

Some Event Subclasses

```
 private class StartSleeping extends Event {
   public StartSleeping(long eventTime) {
     super(eventTime);
   }
   public void action() {
         sleeping = true;
   }
   public String description() {
     return "Buzz off, I'm sleeping!";
   }
}
```

Some Event Subclasses

```
private class StopSleeping extends Event {
  public StopSleeping(long eventTime) {
    super(eventTime);
  }
  public void action() {
        sleeping = false;
  }
  public String description() {
    return "I'm awake now, think I'll study Java.";
  }
}
```

“Bootstrap” Some Events

```
private class AnotherWeekAtCMU extends Event {
    public AnotherWeekAtCMU(long eventTime) {
       super(eventTime);
    }
    public void action() {
       //add event in Controller
       long tm = System.currentTimeMillis();
       addEvent(new StartSleeping(tm));              // Sunday at midnight
       addEvent(new StopSleeping(tm + 28800)); // Monday at 8 am
       addEvent(new StartStudying(tm + 28801));
       addEvent(new StartEating(tm + 28860));
    }
    public String description() {
       return "Starting another week at CMU";
    }
  }
```

Pull It All Together

```
public class LifeAtCMUControls extends Controller {
    private boolean studying = false;
    private boolean sleeping = false;
    private boolean eating = false;
    private boolean playing = false;   // Jeez, what a life!
    // All the previous classes go in here
    public static void main(String[] args) {
        LifeAtCMUControls life = new LifeAtCMUControls();
        long tm = System.currentTimeMillis();
        life.addEvent(life.new AnotherWeekAtCMU(tm));
        life.run();
    }
}
```

Callbacks

- Suppose we want to implement **Incrementable**, yet derive from **MyIncrement**:

```
interface Incrementable {
    void increment();
}

public class MyIncrement {
    public void increment() {
        System.out.println("MyIncrement increment operation");
    }
}
```

A Callee Can “Leave a Message”

```
public class Callee extends MyIncrement {
    private int i = 0;
    private void incr() {
        i++;
        System.out.println("Callee i incremented to " + i);
    }
    private class Closure implements Incrementable {
        public void increment() { incr(); }
    }
    Incrementable getCallbackReference() {
        return new Closure();
    }
}
```

A Caller Can “Call Back”

```
public class Caller {
    private Incrementable callbackReference;
    Caller(Incrementable cbr) {
        callbackReference = cbr;
    }
    void go() {
        callbackReference.increment();
    }
}
```

It Works Like This

```
public class TestCallbacks {
   public static void main(String[] args) {
      Callee c = new Callee();
      c.increment();
      Caller caller = new Caller(c.getCallbackReference());
      caller.go();
      caller.go();
   }
}
```