Lecture 7
Sharing Objects
Publishing and Escape

• *Publishing* an object: making it available to other parts of a program
  – Sometimes you want to
  – Other times you don’t

• Object *escape*: unintended (or poorly considered) publishing
  – Source of many subtle errors
  – Problems can be especially tricky in presence of threads
Perils of Publishing (1/2)

• Consider slight modification to Line class
  
  ```java
  public class BadLine {

      // Invariant: p1 and p2 must be different points
      private MutablePoint p1;
      private MutablePoint p2;

      // BadLine throws exception if points overlap
      BadLine(MutablePoint p1, MutablePoint p2) throws IllegalArgumentException {
          ... error checking ...
      }

      ...
      // get P1() publishes the p1 object
      public class MutablePoint {
          ... private double x;
          ... private double y;

          ...
          public void setX(double newX) { x = newX; }
          public void setY(double newY) { y = newY; }
      }

  }

  • What’s the problem?
      — get P1() publishes the p1 object
      — Code receiving this object can break the line invariant!
Perils of Publishing (2/2)

• What’s the problem?
  – `getP1()` publishes the `p1` field in a `BadLine`
  – Code receiving this object can break the line invariant!
    • Assume `line` is a `BadLine` object
    • What does following code do?
      ```java
      MutablePoint a1 = line.getP1();
      MutablePoint a2 = line.getP2();
      a1.setX(a2.getX());
      a1.setY(a2.getY());
      ```

• This is a problem even in the absence of threads
  – When you publish an object, make sure that that receiving code cannot invalidate invariants
  – Terminology: receiving code sometimes called `alien code` to emphasize this
Obvious Forms of Publishing

• Assigning to a public field
  – Consider
    ```java
    public class ReallyBadLine {
      public MutablePoint p1;
      ...
    }
    – Really bad idea: don’t do this (almost impossible to enforce correctness)

• Via getters (cf. BadLine)
  – Using getters is better than using public fields
  – Remember that once an inner object is obtained by alien code, an enclosing object loses control
Indirect Publishing (1/2)

• Publishing an object also publishes any objects accessible from that object
• Consider (from book)

```java
class UnsafeStates {
    private String[] states = new String[] {
        "AK", "AL", ...
    };

    public String[] getStates() {
        return states;
    }
}
```

– `getStates()` publishes private field `states`, which can now be modified
  (probably not what is intended)
– It also publishes all the `String` objects in the `states` array as well

• **Indirect publishing is the most common form of escape!**
Indirect Publishing (2/2)

• Nested classes can give rise to a subtle form of indirect publishing
  – Inner objects have a reference to outer, enclosing objects
  – This is stored in a hidden field this$0
  – There are means to access this$0
  – So: publishing an inner object indirectly publishes its enclosing object also
Outer / Inner Object Example

• Consider class Outer
  ```java
class Outer {
  private int a = 1;
  public void foo() { System.out.println("Outer a = " + a); }

  public class Inner {
    private int b = a + 1;
    public void foo() { System.out.println("Inner b = " + b); }
  }
}
```

• Now consider (credit to: http://stackoverflow.com/questions/763543/in-java-how-do-i-access-the-outer-class-when-im-not-in-the-inner-class)
  ```java
import java.lang.reflect.Field;

public class OuterInnerTest {
  public static void main(String[] args) {
    Outer.Inner v = new Outer().new Inner();
    v.foo();
    try {
      Field outerThis = v.getClass().getDeclaredField("this$0");
      Outer u = (Outer)outerThis.get(v);
      u.foo();
    } catch (NoSuchFieldException e) { throw new RuntimeException(e); }
    catch (IllegalAccessException e) { throw new RuntimeException(e); }
  }
}
```

• What gets printed?
  ```
  inner b = 2
  Outer a = 1
  Outer object is available, even though it is not directly published
  ```
Multi-Threading and Escape

• Escape is especially problematic in the presence of threads
  – The usual issues of thread-safety are especially evident when an object escapes
  – There is also an issue with incompletely constructed objects being visible to other threads!

• Examples follow
Subtle Escape #1 (1/2)

Here are classes for a collection of cached, time-stamped objects

```java
// Cache class
public class TimeStampedObjCache {
    static public volatile TimeStampedObj lastObjCreated =
        new TimeStampedObj(new Object());
}

// Time-stamped object class
public class TimeStampedObj {
    private final Object payload;
    private final Date timeStamp;

    public TimeStampedObj(Object o) {
        TimeStampedObjCache.lastObjCreated = this;
        this.payload = o;
        timeStamp = new Date();
    }

    public Date getTimeStamp() { return timeStamp; }

    public Object getPayload() { return payload; }
}
```
Subtle Escape #1 (2/2)

- What will this driver do?
  ```java
  public static void main(String[] args) {
      int errorCount = 0;
      int iterations = 10000;
      Thread t1;

      for (int i=0; i<iterations; i++) {
          t1 = new Thread(new Runnable() {
              public void run() { new TimeStampedObj(new Object()); }
          });
          t1.start();
          if (TimeStampedObjCache.lastObjCreated.getTimeStamp() == null) {
              errorCount++;
          }
      }
      System.out.println(errorCount);
  }
  ```

- It seems like the error count should be 0, and yet on most architectures it is not!
  - Some TimeStampedObj objects are not fully constructed when they are assigned to cache
  - When getDate() is called on them, they can return null!
Subtle Escape #2

  ```java
  public class EventListener {
      public EventListener(EventSource eventSource) { eventSource.registerListener(this); } 
      public onEvent(Event e) { } 
  }
  public class RecordingEventListener extends EventListener {
      private final ArrayList list;
      public RecordingEventListener(EventSource eventSource) {
          super(eventSource); //HAS TO BE FIRST LINE OF CONSTRUCTOR
          list = Collections.synchronizedList(new ArrayList());
      }
      public onEvent(Event e) {
          list.add(e);
          super.onEvent(e);
      }
      public Event[] getEvents() { return (Event[]) list.toArray(new Event[0]); } 
  }
  ```

- this is published in EventListener constructor
- Any thread with access to eventSource listeners now has access to this object
- RecordingEventListener now extends EventListener
  - Constructor is also extended
  - RecordingEventListener objects can be accessible even before list is added to object!
Morals

• Object is only fully constructed when constructor terminates
• Don’t let \texttt{this} escape during object construction!
  – Don’t do it!
  – Book: object is \textit{improperly constructed} when this is the case)

• Related point
  – Don’t start threads inside constructors
  – Reason: very easy to publish \texttt{this} to such threads
A Safe Construction Paradigm

• In subtle escape examples, problem stemmed from desire to publish object as part of its creation
  – In #1, `TimeStampedObj` objects assigned to cache as part of construction
  – In #2, `EventListener` objects registered with event sources

• Desire to do this is understandable!
  – Key functionality for these objects is to be part of these larger objects
  – Problem is that object is not fully constructed until after constructor terminates
  – In multi-threaded systems, a thread might see an incompletely constructed object

• We can achieve this using private constructors and a static `factory method`
  – New method acts as “proxy” for constructing objects, installing them properly
  – Method calls private constructor, then installs it in appropriate data structures
public class FixedTimeStampedObj {
    private final Object payload;
    private final Date timeStamp;

    // To avoid publishing this in constructor, make it private and
    // do not assign to cache
    private FixedTimeStampedObj(Object o) {
        this.payload = o;
        this.timeStamp = new Date();
    }

    // Static factory method is what users use to create objects now
    public static FixedTimeStampedObj newInstance(Object o) {
        FixedTimeStampedObj tso = new FixedTimeStampedObj(o);
        FixedTimeStampedObjCache.lastObjCreated = tso;
        return tso;
    }

    ...

Thread Confinement

• Sharing objects among threads imposes costs
  – Thread-safety must be implemented explicitly
  – This involves locking
  – Locking incurs run-time overhead, programming complexity

• One way to minimize complexity: don’t share!
  – Of course, some sharing is needed
  – However, objects that are confined to a single thread are guaranteed to be thread-safe
  – Many graphical-user-interface (GUI) follow this paradigm
    • There is a single thread handling events
    • Applications put events into event queue
    • Handler repeatedly checks event queue, calls appropriate handler
    • Objects that only reside in handler need not be synchronized
Ad hoc Thread Confinement

• Programmer uses her / his ingenuity to ensure thread confinement

• One common paradigm
  – When you create a new thread, give it its own deep copy of the local objects it needs
  – These local objects will be thread-confined

• Dangers!
  – Frequently, only programmer knows about this design goal
  – It’s easy to make mistakes
  – Document!
Stack Confinement

• Local variables belong to a single thread, by definition
  – Local variables live on the stack
  – In Java, only the heap is shared

• Objects will be *stack confined* if they are:
  – Created in a thread
  – Assigned to a local variable in the thread
  – Never published