Design of Thread-Safe Classes

Topic Outline

- Thread-Safe Classes Principles
- Confinement
- Delegation
- Synchronization policy documentation

Thread-safe Class Design Process

- Identify the object's state (variables)
- Identify state invariants
- Establish policy for managing concurrent access to the object's state

Give examples of three types of constraints to consider when identifying state invariants.

You must understand an object's invariants during the design process to ensure thread safety.

Thread-safe: Post-conditions

- Post-conditions – what transformation does a method perform?
  - New start in terms of original state.
  - Example: set_v1 postcondition is v2 and v3 unchanged.
  - Related to but different from invariant – defines legal ***state transitions***.

MORAL

It is impossible to ensure thread safety without knowledge of the object's state invariant and its method post-conditions. Such constraints generate atomicity and encapsulation requirements.

Thread-safe: State Dependent Operations

- ***Post-conditions*** – what transformation does a method perform?
- ***Pre-conditions*** – under what conditions can a method be called?
  - Example: Try to pop an element off an empty stack.
  - In sequential system this is an error:
    - Return **null** value.
    - Raise an exception.
  - With concurrency, may be a temporary condition - design choice.
- Handle with stack's lock + wait / notify.
  - Potential problem: Tightly coupled to the intrinsic locking mechanism.
  - With richer structures, can be difficult to get correct.
  - Better solution: Use of library *synchronizer* classes (next chapter).

Thread-safe: State Ownership

- Class design element : data that object owns
- Owner of the object decides locking protocol
  - Be aware if/when object gets published
  - Be aware of object references within objects that escape
- Need to think about ownership models when passing objects:
  - Are you transferring ownership?
  - Making a short-term loan?
  - Creating a joint-ownership?
  - Split ownership.

What is an example of the split ownership model.

Thread-safe Classes: State

- Start with the fields:
  - All primitive – that's it!
  - Object references – include those object's fields.
- State is whatever data determines object structural integrity

```
public class Point {
  private float x, y ;

  public Point(float x, float y) {
    this.x = x ; this.y = y ;
  }

  public float get_x() { … }
  public float get_y() { …}

  public float set_x(float nx) { … }
  public float set_y(float ny) { … }
}
```

```
public class Triangle {
  private Point v1, v2, v3 ;

  public Triangle(Point v1, Point v2, Point v3) {
    . . .
  }
}
```

Thread-safe Classes: Invariant

- Invariant = structural rules of the road.
- For a triangle, points cannot be *collinear*.
- Where can Triangle invariant be violated?

```
public class Point {
  private float x, y ;

  public Point(float x, float y) {
    this.x = x ; this.y = y ;
  }

  public float get_x() { … }
  public float get_y() { …}

  public float set_x(float nx) { … }
  public float set_y(float ny) { … }
}
```

```
public class Triangle {
  private Point v1, v2, v3 ;

  public Triangle(Point v1, Point v2, Point v3) {
    if ( colinear(v1,v2,v3)  )
      throw new Error() ;
    this.v1 = v1 ; this.v2 = v2 ; this.v3 = v3 ;
  }

  public void set_v1(Point nv1) {
    if (colinear(nv1, v2, v3) )
      throw new Error() ;
    v1 = nv1 ;
  }

  public void set_v2(Point nv2) { … }

  public void set_v3(Point nv3) { … }

  public Point[] get_vertices( )  { … }

  private boolean colinear(Point v1, Point v2, Point v3) { … }
}
```

Thread-safe Classes: Invariant

- Fix - make Points immutable
- Is this sufficient?

```
public class Point {
  private final float x, y ;

  public Point(float x, float y) {
    this.x = x ; this.y = y ;
  }

  public float get_x() { … }
  public float get_y() { …}

  public float set_x(float nx) { … }
  public float set_y(float ny) { … }
}
```

```
public class Triangle {
  private Point v1, v2, v3 ;

  public Triangle(Point v1, Point v2, Point v3) {
    if ( colinear(v1,v2,v3)  )
      throw new Error() ;
    this.v1 = v1 ; this.v2 = v2 ; this.v3 = v3 ;
  }

  public void set_v1(Point nv1) {
    if (colinear(nv1, v2, v3) )
      throw new Error() ;
    v1 = nv1 ;
  }

  public void set_v2(Point nv2) { … }

  public void set_v3(Point nv3) { … }

  public Point[] get_vertices( )  { … }

  private boolean colinear(Point v1, Point v2, Point v3) { … }
}
```

Thread-safe Classes: Concurrent Access

- Fix - make Points immutable
- Is this sufficient?

```
public class Point {
  private final float x, y ;

  public Point(float x, float y) {
    this.x = x ; this.y = y ;
  }

  public float get_x() { … }
  public float get_y() { …}
}
```

```
public class Triangle {
  private Point v1, v2, v3 ;

  public Triangle(Point v1, Point v2, Point v3) {
    if ( colinear(v1,v2,v3)  )
      throw new Error() ;
    this.v1 = v1 ; this.v2 = v2 ; this.v3 = v3 ;
  }

  public void synchronized set_v1(Point nv1) {
    if (colinear(nv1, v2, v3) )
      throw new Error() ;
    v1 = nv1 ;
  }

  public void synchronized set_v2(Point nv2) { … }

  public void synchronized set_v3(Point nv3) { … }

  public Point[] synchronized get_vertices( )  { … }

  private boolean colinear(Point v1, Point v2, Point v3) { … }
}
```

Topic Outline

- Design of Thread-Safe Classes
- Confinement
- Delegation
- Synchronization policy documentation

(Instance) Confinement

- Want to use thread unsafe objects in a thread safe way.
- Encapsulate in wrapper object.
  - Encapsulation makes it easier to ensure proper lock held when manipulating the wrapped object.
  - Proxy pattern example:
    - Same interface - just controlled for concurrency
    - Synchronized methods in proxy interface
    - Explicit synchronization on unsafe object's lock
  - Or wrapper could be abstraction built on unsafe class
    - Thread safe course with enrollment based on unsafe HashSet.
    - Thread safe queue based on unsafe ArrayList.

What design pattern is used & how might it support safe access?

- Java "monitor" pattern.
- Note - can be the default lock or any other one used consistently.

Confinement Example - Implicit Lock

```
public class Dictionary {
  private final HashMap<String, String> dict =
    new HashMap<String, String>() ;

  public synchronized String getDef(String word) {
    return dict.get(word) ;
  }

  public synchronized void addDef(String word, String def) {
    if ( dict.get(word) == null ) {
      dict.put( word, def ) ;
    }
  }

  public synchronized Map<String, String> getMap() {
    return Collections.unmodifiableMap( dict ) ;
  }
}
```

What is being used as the lock in this example?

Confinement Example - Explicit Lock

```
public class Dictionary {
  private final HashMap<String, String> dict =
    new HashMap<String, String>() ;

  private Object lock = new Object() ;

  public String getDef(String word) {
    synchronized(lock) {
      return dict.get(word) ;
    }
  }

  public synchronized void addDef(String word, String def) {
    synchronized(lock) {
      if ( dict.get(word) == null ) {
        dict.put( word, def ) ;
      }
    }
  }

  public synchronized Map<String, String> getMap() {
    synchronized(lock) {
      return Collections.unmodifiableMap( dict ) ;
    }
  }
}
```

Is there an advantage to using a explicit vs an implicit lock?

Topic Outline

- Design of Thread-Safe Classes
- Confinement
- Delegation
- Synchronization policy documentation

Delegation

- Using thread safe classes "under the hood."
- Can sometimes delegate safety to these classes.

```
public class Dictionary {
  private final ConcurrentMap<String, String> dict =
    new ConcurrentHashMap<String, String>() ;
  private final Map<String, String> unmodifiable =
    Collections.unmodifiableMap( dict ) ;

  public String getDef(String word) {
    return dict.get(word) ;
  }

  public void addDef(String word, String def) {
    dict.putIfAbsent( word, def ) ;
  }

  public Map<String, String> getMap() {
    return unmodifiable ;
  }
}
```

Delegation Failure

```
public class Range{
  // Invariant: min <= max
  // AtomicInteger provides for thread-safe min & max objects
  private final AtomicInteger min = new AtomicInteger(0) ;
  private final AtomicInteger max = new AtomicInteger(0) ;

  public void setMin(int m) {
    if ( m > max.get() )
      throw new Error("min " + m + " > max") ;
    min.set(m) ;
  }

  public void setMax(int m) {
    if ( m > max.get() )
      throw new Error("max " + m + " < min") ;
    max.set(m) ;
  }

  public boolean inrange(int i) {
    return i >= min.get() && i <= max.get() ;
  }
}
```

Is Range thread-safe?

Why or why not?

Topic Outline

- Design of Thread-Safe Classes
- Confinement
- Delegation
- Synchronization policy documentation

Synchronization Policy Documentation

- Clients need to know thread safety guarantees if any.
- Maintainers / extenders need to know synchronization policy.
  - Use of immutable objects.
  - Use of volatile variables.
  - Use of synchronized to create thread safe objects.
  - Use of delegation to create thread safe objects.
- Useful information not found in the code
  - What invariant interdependencies exist (e.g., iteration)
  - How interdependencies are guarded.
  - What each lock controls (scope).

Looking ahead…

- We have covered approaches to concurrent design that primarily use “traditional” Java mechanisms to deal with mutable state.
- Three options for addressing state in design:
  1. Shared Mutability
  2. Isolated Mutability
  3. Immutability
- ***Design with immutability*** requires more effort since it requires a different design approach than we may be used to.
- …But, the tradeoff is **LESS** need for low level synchronization mechanisms on our part and fewer concurrency headaches.