Immutability

Lecture 8

Vocabulary: Accessors & Mutators

- **Accessor:**
  - A method that reads, but never changes, the (abstract) state of an object
  - Concrete representation may change, so long as change is not visible to client
  - Examples: getter methods, toString
  - Formally: Alters clause does not include "this"
  - recall RESOLVE functions

- **Mutator:**
  - A method that may change the (abstract) state of an object
  - Examples: setter methods
  - Formally: Alters clause includes "this"
  - recall RESOLVE procedures
  - Constructors not considered mutators

An Epoch Interface

```java
public interface Epoch {
    // Returns: beginning
    public Date getStart();
    // Returns: ending
    public Date getEnd();
    // Requires: factor >= 0
    // Alters: this.ending
    // Ensures: length = (1 + #factor) * #length
    public void stretch(float factor);
}
```

Questions

- **What is an invariant in general?**
  - Ans:

- **What is an invariant for Epoch?**
  - Ans:

- **Why is this an invariant?**
  - Ans:

Relying on an Invariant

```java
public class CreditCard {
    BigDecimal interest (BigDecimal balance, Epoch e) {
        long msTime = e.getEnd().getTime();
        msTime = msTime - e.getStart().getTime();
        assert (msTime > 0); //always true
        ... //code to calculate interest on balance
    }
    BigDecimal forgiveness (Epoch e) {
        Date oldDueDate = e.getEnd();
        e.stretch(0.3);
        assert (oldDueDate.compareTo(e.getEnd()) < 0);
        //oldDueDate < e.ending, always true
    }
}
```

A Fixed Epoch Interface

```java
public interface FixedEpoch {
    // Returns: beginning
    public Date getStart();
    // Returns: ending
    public Date getEnd();
}
```

```java
public class CreditCard {
    BigDecimal interest (BigDecimal balance, Epoch e) {
        long msTime = e.getEnd().getTime();
        msTime = msTime - e.getStart().getTime();
        assert (msTime > 0); //always true
        ... //code to calculate interest on balance
    }
    BigDecimal forgiveness (Epoch e) {
        Date oldDueDate = e.getEnd();
        e.stretch(0.3);
        assert (oldDueDate.compareTo(e.getEnd()) < 0);
        //oldDueDate < e.ending, always true
    }
}
```
A Broken Time Period Class

```java
public class Period implements FixedEpoch {
    private Date start;
    private Date end;
    public Period(Date start, Date end) {
        assert (start.compareTo(end) < 0); // start < end
        this.start = start;
        this.end = end;
    }
    public Date getStart() {
        return start;
    }
    public Date getEnd() {
        return end;
    }
}
```

Problem: Aliasing

- Assignment in constructor creates an alias
  - Client and component both have references to the same Date object
- Class invariant can be undermined via alias
  
  ```java
  Date t1 = new Date(300);
  Date t2 = new Date(500);
  Period p = new Period(t1, t2);
  t2.setTime(100); // modifies p's rep
  ```
- Solution: "defensive copying"
  - Constructor creates a copy of the arguments
  - Copy is used to initialize the private fields
  - Metaphor: ownership

A Better Period Class

```java
public class Period implements FixedEpoch {
    private Date start;
    private Date end;
    public Period(Date start, Date end) {
        this.start = new Date(start.getTime());
        this.end = new Date(end.getTime());
        assert (this.start.compareTo(this.end) < 0);
    }
    public Date getStart() {
        return start;
    }
    public Date getEnd() {
        return end;
    }
}
```

Good Practice: Copy First

- When making a defensive copy of constructor arguments:
  - First copy the arguments
  - Then check the validity of the parameters
- Reason: multithreaded code
  - Consider a constructor that checks first, then copies
  - Another thread of execution could change the parameters after they pass the validity check, but before they are copied into the private fields

Problem 2: Aliasing (Again)

- Return value in accessor creates an alias
  - Client can still obtain a reference to the class's internal representation (the private fields)
  - aka "privacy leak", but really just an alias problem
- Class invariant can be undermined via alias
  
  ```java
  Date t1 = new Date(300);
  Date t2 = new Date(500);
  Period p = new Period(t1, t2);
  p.getEnd().setTime(100); // modifies p's rep
  ```
- Solution: "defensive copying"
  - Accessors create a copy of internal fields
  - Copy is returned to the client

A Better+1 Period Class

```java
public class Period implements FixedEpoch {
    private Date start;
    private Date end;
    public Period(Date start, Date end) {
        this.start = new Date(start.getTime());
        this.end = new Date(end.getTime());
        assert (this.start.compareTo(this.end) < 0);
    }
    public Date getStart() {
        return start;
    }
    public Date getEnd() {
        return end;
    }
}
```
A Better+2 Period Class

public class Period implements FixedEpoch {
    private Date start;
    private Date end;
    
    public Period(Date start, Date end) {
        this.start = new Date(start.getTime());
        this.end = new Date(end.getTime());
        assert (this.start.compareTo(this.end) < 0);
    }
    
    public Date getStart() {
        return new Date(start.getTime());
    }
    
    public Date getEnd() {
        return new Date(end.getTime());
    }
}

Good Practice: Defensive Copies

- Always make defensive copies when needed
  - Problem: Aliases undermine the privacy of a field
  - Solution: Prevent aliases to fields
- Typical examples
  - Parameters in constructors and mutators
  - Return value from any method
- Note: There are some types of fields for which aliasing is never a concern!
  - Fields that are primitive (e.g., int, float)
  - Fields that are enumerations (e.g., Suit, Colors)
  - Fields that are...

Immutability

- An immutable object is one whose (abstract) value can never change
  - Constructor allows initialization to different values
  - No mutator methods
- Why would we want such a thing?
  - Because aliasing an immutable is safe!
  - Having multiple references to the same immutable is indistinguishable from having multiple references to different immutables that have the same value
  - Defensive copies of immutables are not required!

How to Write an Immutable Interf.

- Do not provide mutators
  - Check alters clause of all methods

How to Write an Immutable Class

- Implement an (immutable) interface
  - Result: no mutators
  - You do that anyway, right?
- Make all fields private
  - You do that anyway, right?
- Ensure exclusive access to any mutable objects referred to by fields
  - Rule: If the class has fields that refer to mutable objects
    1. Make defensive copies of parameters in constructors and methods
    2. Make defensive copies for return values from methods
  - Defensive copies not needed for fields that are primitive, enumerations, or refer to immutable objects

Examples

- Period
  - Has fields that refer to mutables (Date)
  - Needs defensive copies
- String
  - Lots of methods look like they could be mutators
    - eg toUpperCase(), substring(int, int), replace(char, char)
  - But these methods actually return a String
    - String str = new String("Hello there");
    - str.toUpperCase();
    - System.out.println(str); //surprise
- Wrapper classes
  - Integer, Long, Float, etc..
Good Practice: Immutable Idioms

- Declare all fields to be final
  - Guarantees immutability for primitives
  - Underkill: For reference types, final is no help
    - Still considered an idiom that signals intent to write an immutable class
  - Overkill: Only abstract state needs to be immutable
    - Concrete state (ie fields) can change so long as client-view of object is unchanged
- Declare class to be “final”
  - We will talk about what this qualifier means for classes later

A Better+3 Period Class

```java
public final class Period implements FixedEpoch {
  private final Date start;
  private final Date end;
  public Period(Date start, Date end) {
    this.start = new Date(start.getTime());
    this.end = new Date(end.getTime());
    assert (this.start.compareTo(this.end) < 0);
  }
  public Date getStart() {
    return new Date(start.getTime());
  }
  public Date getEnd() {
    return new Date(end.getTime());
  }
}
```

Wrapper Classes

- Every primitive type has a corresponding wrapper class
  - Integer, Long, Float, Double, ...
- The classes are immutable
  - So no aliasing worries
- Do not provide a zero-argument constructor
- Do provide useful static constants
  - Integer.MAX_VALUE, Integer.MIN_VALUE
- Do provide useful static methods
  - Converting from String to primitive: parseInt()
    - `int i = Integer.parseInt("33342");`
  - Converting from primitive to String: toString()
    - `String str = Double.toString(123.99);`

Boxing and Unboxing

- Boxing: primitive --> wrapper
  ```java
  Integer integerObject = new Integer(42);
  ```
- Unboxing: wrapper --> primitive
  ```java
  int i = integerObject.intValue();
  ```
- Java does this automatically for you
  ```java
  Double price = 499.99; //auto-box
  price = price + 19.90; //auto-unbox then box
  ```
- But be very careful...
  ```java
  Integer i = new Integer(2);
  Integer j = new Integer(2);
  assert (i >= j); //success (unboxing)
  assert (i <= j); //success (unboxing)
  assert (i == j); //Failure! (no unboxing)
```

Supplemental Reading

- Bloch’s “Effective Java”
  - Item 15: Minimize mutability
  - Item 39: Make defensive copies when needed
- IBM developerWorks paper
  - "Java theory and practice: To mutate or not to mutate?"

Summary

- Defensive copying
  - Copy constructor arguments (reference types)
  - Return only copies of fields (reference types)
- Immutable interfaces and classes
  - Each instance represents a distinct value
  - No mutators: no methods alter “this”
  - Methods can return a new instance
  - Defensive copying of mutable fields
- Examples of immutables
  - String
  - Wrapper classes (Integer, Long, Float...)
- Auto-boxing / auto-unboxing