SWE 619 Course Schedule and Assignments

ThanhVu (Vu) Nguyen

October 18, 2021

Contents

1 Schedule

- **Notes**: Except for the first class, students are expected to complete the reading **prior to class** on listed meeting. Quizzes may occasionally take advantage of this expectation.

  Homework assignments are **due** on the date listed. Quizzes are posted, with solutions, on Piazza following the class in which the quiz is given.

Meeting	Topic	Reading	Assignme
1	Class Overview	Liskov 1–2	
2	Procedural Abstraction; Exceptions	Liskov 3–4, Bloch 10	A1 Du
3	Data Abstraction; Mutability	Liskov 5.1–5.4,	A2 Du
4-1	Program Verification	Hoare logic notes	A4-1 Du
4	Reasoning About Data Abstraction	Liskov 5.5-5.10	A3 Du
5	Interation Abstraction; Method Guidelines	Liskov 6, Bloch 8	A4 Du
6	Type Abstraction	Liskov 7	A5 Du
7	Mid-semester Recap	*No New Material*	G
8	Polymorphic Abstract; Lambdas	Liskov 8, Bloch 7(Item 42)	A6 Du
9	Generics	Bloch 5	A7 Du
10	Common Java Contracts	Bloch 3	A8 Du
11	Classes and Inheritance	Bloch 4	A9 Du
12	Contract Model in Testing	Advanced JUnit (slides 22-24)	A10 Du
13	Enums and Annotations	Bloch 6	A11 Du
14	Course Wrap Up	Final Exam Notes; Code for final	
15	Final Exam @ 4:30	*Online*	

2 In class Assignments

2.1 In class 1

Work with your group and do the following:

1. Spend a few minutes getting acquainted. Explain a bit about yourself: full-time student?, working in software development?, why are you taking this class?, favorite/least favorite thing about writing software?, etc.

2. Decide on a mechanism for joint communication. Google docs? IDE with screen share? Something else?

Now address a technical topic. This exercise touches on some of the thorny issues in data abstraction and inheritance. There is a lot going on in this example. Hence don't worry if it seems confusing today. We'll revisit this example several times over the course of the semester.

Consider the following (textbook) code:

```
public class User {
    private String name;
```

```
    public User (String name) { this.name = name; }
    @Override public boolean equals (Object obj) {
        if (!(obj instanceof User)) return false;
        return ((User) obj).name.equals(this.name);
    }
    // other methods omitted
}
public class SpecialUser extends User {
    private int id;
    public SpecialUser (String name, int id) { super(name); this.id = id; }
    @Override public boolean equals (Object obj) {
        if (!(obj instanceof SpecialUser)) return false;
        return super.equals(obj) && ((SpecialUser) obj).id == this.id;
    }
    // other methods omitted
}
```

1. Walk though the execution of the `equals()` method in class `User` for a few well-chosen objects as the parameter. What happens at each point in the execution?
2. What does it mean for an `equals()` implementation to be **correct**? How do you know? Be as concrete as you can.
3. Is the given implementation of `equals()` in class `User` correct? Again, be concrete. If there is a problem, find a specific object (test case!) that demonstrates the problem.
4. How does inheritance complicate the correctness discussion for `equals()` in class `SpecialUser`?
5. What is your assessment of the `equals()` method in the `SpecialUser` class?

2.2 In class 1A

Consider the following specification and implementation:

```
public static List<Integer> tail (List<Integer> list) {

    // REQUIRES: ???
```

```
    // EFFECTS:  ???

    List<Integer> result = new ArrayList<Integer>(list);
    result.remove(0);
    return result;
}
```

1. What does the **implementation** of `tail` do in each of the following cases? How do you know: Running the code or reading an API description?
   - `list = null`
   - `list = []`
   - `list = [1]`
   - `list = [1, 2, 3]`
2. Write a partial specification that matches the "happy path" part of the implementation's behavior.
3. Rewrite the specification to be total. Use Bloch's standard exceptions.
4. The resulting specification has a problem. What is it?
5. Rewrite the specification to address this problem. Rewrite the code to match the new specification.

2.3 In class 1B

Goal: Understanding Contracts

Consider the 3 methods `hasNext` , `next`, and `remove` in the Java Iterator interface:

- For each method, identify all preconditions and postconditions.
- For each precondition, identify a specific input that violates the precondition.
- For each postcondition, identify an input specific to that postcondition.

2.4 In class 2A

Consider a simple generic `Queue` implementation.

```
public class Queue <E> {

    private List<E> elements;
    private int size;

    public Queue() {
        this.elements = new ArrayList<E>();
        this.size = 0;
    }

    public void enQueue (E e) {
        elements.add(e);
        size++;
    }

    public E deQueue () {
        if (size == 0) throw new IllegalStateException("Queue.deQueue");
        E result = elements.get(0);
        elements.remove(0);
        size--;
        return result;
    }

    public boolean isEmpty() {
        return size == 0;
    }
}
```

1. Rewrite `Queue` to be **immutable**. Keep the representation variables `elements` and `size`.
2. Do the right thing with `enQueue()`.
3. Do the right thing with `deQueue()`.

2.5 In class 2B

Consider Liskov's Poly example, where an abstract Poly is defined as $c_0 + c_1x + c_2x^2 + \ldots$, and is implemented with two variables:

```
private int deg;
private int[] trms;

Fill in example values that are mapped by the abstraction function.
Abstract Poly State:
What is a "state"?



AF
/|\
 |
 |
 |
 |-----------------------------------------------------------
 |
 |
 |
 |
```

```
Representation State: (deg, trms)
```

1. Identify representation states that should not be mapped.
2. Try to capture these states with a rule (that is, a rep-invariant).
3. Devise a representation that is suitable for a mutable version of Poly.
4. Develop a rep-invariant for that representation.

2.6 In class 3A

Consider Liskov's immutable `Poly` example, where an abstract `Poly` is defined as $c_0 + c_1x + c_2x^2 + \ldots$, and is implemented with one variable:

```
private Map<Integer, Integer> map;
```

Fill in example values that are mapped by the abstraction function.

```
Abstract State: Poly

AF
/|\
|
|
|
|-----------------------------------------------------------
|
|
|
|
```

```
Representation State: map
```

1. Identify representation states that should not be mapped.
2. Try to capture these states with a rule (that is, a rep-invariant).
3. Consider implementing the `degree()` method. What code would do the job? What more specific type of map would make the implementation simpler?

2.7 In class 3B

Consider the code:

```
public class Members {
```

```
// Members is a mutable record of organization membership
// AF: Collect the list as a set
// rep-inv1: members != null
// rep-inv2: members != null && no duplicates in members
// for simplicity, assume null can be a member...

List<Person> members;   // the representation

//  Post: person becomes a member
public void join (Person person) { members.add   (person);}

//  Post: person is no longer a member
public void leave(Person person) { members.remove(person);}
```

1. Analyze these 4 questions for rep-inv 1.
   (a) Does `join()` maintain rep-inv?
   (b) Does `join()` satisfy contract?
   (c) Does `leave()` maintain rep-inv?
   (d) Does `leave()` satisfy contract?
2. Repeat for rep-inv 2.
3. Recode `join()` to make the verification go through. Which rep-invariant do you use?
4. Recode `leave()` to make the verification go through. Which rep-invariant do you use?

2.8 In class 4-1

```
// {N >= 0}   # P
i = 0;
while (i < N){
  i = i + 1;
}

//{i == N}  # Q
```

- Identify the loop invariants for the loop in this program

- Use a sufficiently strong invariant to prove the program is correct
- Attemp to prove the program using an insufficiently strong invariant, describe what happens and why.

2.9 In class 4A

Consider the Java `Iterator<E>` interface:

```
public boolean hasNext();
public E next() throws NoSuchElementException
                       public void remove() throws IllegalStateException
```

1. What is the abstract state of an iterator without the `remove()` method?
2. Work through an example iterating over a list of strings: `["bat", "cat", "dog"]`
3. What is the abstract state of an iterator with a `previous()` method?
4. What is the abstract state of an iterator with the `remove()` method?
5. Design an immutable version of the iterator.
   (a) How is `hasNext()` handled?
   (b) How is `next()` handled?
   (c) How is `remove()` handled?
6. Exercise the immutable iterator with some sample client code.

2.10 In class 4B

Consider the example in Bloch's Item 50 (3rd Edition):

```
// Broken ‘‘immutable’’ time period class
public final class Period {               // Question 3
    private final Date start;
    private final Date end;

    /**
     * @param start the beginning of the period
     * @param end the end of the period; must not precede start
```

```
     * @throws IAE if start is after end
     * @throws NPE if start or end null
     */

    public Period (Date start, Date end) {
        if (start.compareTo(end) > 0) throw new IAE();
        this.start = start; this.end = end;  // Question 1
    }
    public Date start() { return start;}    // Question 2
    public Date end()   { return end;}      // Question 2
}
```

1. Write code that shows the problem the line marked // Question 1.
2. Write code that shows the problem the lines marked // Question 2.
3. Suppose that the class declaration were:

   ```
   public class Period { // Question 3
   ```

   - Write code that shows the problem.

4. Bloch fixes the constructor as follows:

   ```
   public Period (Date start, Date end) {
       this.start = new Date(start.getTime());  // Defensive copy
       this.end   = new Date(end.getTime());    // Defensive copy

       if (this.start.compareTo(end) > 0) throw new IAE();
   ```

   (a) Bloch states that `clone()` would be inappropriate for copying the dates. Write code that shows the problem.

   (b) Bloch defers the exception check until the end, which seems to violate normal practice. What's the problem with checking early?

2.11 In class 5A

Goal: Understanding dynamic dispatching

Consider Liskov's `MaxIntSet` example with explict `repOk()` calls: (Really, we'd need assertions on these calls...)

```
public class IntSet {
    public void insert(int x) {...; repOk();}
    public void remove(int x) {...; repOk();}
    public boolean repOk() {...}
}
public class MaxIntSet extends IntSet {
    public void insert(int x) {...; super.insert(x); repOk();}
    public void remove(int x) {super.remove(x); ...; repOk();}
    public boolean repOk() {super.repOk(); ...;}
}

MaxIntSet s = {3, 5}; s.remove(5);  // repOk()????
```

1. What do the `"..."` bits do?
2. How does the call work out?
3. What is the abstract state of a `MaxIntSet`? There are two options. What are they, and what are the consequences of each choice?

2.12 In class 5B

Consider the following:

```
class A:
    public void reduce (Reducer x)
        // Effects: if x is null throw NPE
        // else if x is not appropriate for this throw IAE
        // else reduce this by x

class B:
    public void reduce (Reducer x)
        // Requires: x is not null
        // Effects: if x is not appropriate for this throw IAE
        // else reduce this by x

class C:
    public void reduce (Reducer x)
        // Effects: if x is null return (normally) with no change to this
```

```
        // else if x is not appropriate for this throw IAE
        // else reduce this by x
```

Analyze the "methods rule" for `reduce()` in each of these cases: Note: Some analysis may not be necessary. If so, indicate that.

```
B extends A.
Precondition Part:
Postcondition Part:
-----------------------------------
C extends A.
Precondition Part:
Postcondition Part:
-----------------------------------
A extends B.
Precondition Part:
Postcondition Part:
-----------------------------------
C extends B.
Precondition Part:
Postcondition Part:
-----------------------------------
A extends C.
Precondition Part:
Postcondition Part:
-----------------------------------
```

2.13 In class 5C

Consider the following:

```
public class Counter{   // Liskov 7.8
    public Counter()     //EFF: Makes this contain 0
        public int get()     //EFF: Returns the value of this
        public void incr()   //MOD: this //EFF: makes this larger
        }
public class Counter2 extends Counter { // Liskov 7.9
    public Counter2()         //EFF: Makes this contain 0
        public void incr()       // MOD: this //EFF: double this
        }
```

```
public class Counter3 extends Counter {  // Liskov 7.10
    public Counter3(int n)   //EFF: Makes this contain n
        public void incr(int n)  // MOD: this //EFF: if n>0 add n to this
        }
```

1. Is there a constraint about negative/zero values for this? How do we know?
2. What methods are in the `Counter2` API?
3. Is `Counter2` a valid subtype of Counter?
4. What methods are in the `Counter3` API?

2.14 In class 6

This is a recap exercise.

```
public class BoundedQueue {
    private Object rep[];
    private int front = 0;
    private int back = -1;
    private int size = 0;
    private int count = 0;

    public BoundedQueue(int size) {
        if (size > 0) {
            this.size = size;
            rep = new Object[size];
            back = size - 1;
        }  }

    public boolean isEmpty() { return (count == 0); }
    public boolean isFull() { return (count == size); }
    public int getCount() { return count; }

    public void put(Object e) {
        if (e != null && !isFull()) {
            back++;
            if (back >= size)
                back = 0;
```

```
            rep[back] = e;
            count++;
        } }

    public Object get() {
        Object result = null;
        if (!isEmpty()) {
            result = rep[front];
            rep[front] = null;
            front++;
            if (front >= size)
                front = 0;
            count--;
        }
        return result;
    }
    @Override public String toString() {
        String result = "front = " + front;
        result += "; back = " + back;
        result += "; size = " + size;
        result += "; count = " + count;
        result += "; rep = [";
        for (int i = 0; i < rep.length; i++) {
            if (i < rep.length-1)
                result = result + rep[i] + ", ";
            else
                result = result + rep[i];
        }
        return result + "]";
    }
}
```

1. What is wrong with `toString()`? What needs to be done to fix it? Make it so.
2. Write some sample client code to exercise the data structure. Include some non-happy-path cases.
3. Write contracts for each method (as written), including the constructor.

4. Build a rep-invariant. Focus on the code in `get()`. There are also lots of constraints on the array indices; these are quite tricky to get right. The constructor also introduces some complexity.

5. Suppose we removed the line

   ```
   rep[front] = null;
   ```

   from `get()`.

   (a) Informally, why is this wrong?

   (b) Formally, where does the correctness proof break down?

   (c) Could a client ever see the problem?

6. Now that we've done some AF/RI analysis, what changes make the implementation better? btw - this is code straight out of a textbook.

7. Could this data structure be made immutable? If so, what would change in the contracts and method headers? What would likely change in the implementation?

2.15 In class 7

Lambda In class; Comparator; Contract;

2.16 In class 8A

Given the following variable declarations, independently consider the given 6 sequences of Java instructions.

```
String           string = "bat";
Integer          x = 7;
Object[]         objects;
List             rawList;
List < Object >  objectList;
List < String >  stringList;
```

Identify any code that results in a compiler error or warning. Identify any code that raises a runtime exception. Once a compiler error is noted, you do not need to analyze the sequence further.

1. ```
   objects = new String[1];
   objects[0] = string;
   objects[0] = x;
   ```

2. ```
   objects = new Object[1];
   objects[0] = string;
   objects[0] = x;
   ```

3. ```
   stringList = new ArrayList < String >();
   stringList.add(string) ;
   ```

4. ```
   objectList = new ArrayList < String >();
   objectList.add(string) ;
   ```

5. ```
   objectList = new ArrayList < Object >();
   objectList.add(string) ;
   objectList.add(x) ;
   ```

6. ```
   rawList = new ArrayList();
   rawList.add(string) ;
   rawList.add(x) ;
   ```

2.17 In class 8B

```
// Chooser - a class badly in need of generics!
// Bloch 3rd edition, Chapter 5, Item 28:  Prefer lists to arrays

public class Chooser {
    private final Object[] choiceArray;

    public Chooser (Collection choices) {
        choiceArray = choices.toArray();
    }

    public Object choose() {
        Random rnd = ThreadLocalRandom.current();
        return choiceArray [rnd.nextInt(choiceArray.length)];
    }
}
```

- First, simply generify by adding a type to the Chooser class. What is the compiler error with this approach?

- How can you turn the compiler error into a compiler warning?
- Can this warning be suppressed? Should it?
- How can you adopt Bloch's advice about arrays and lists to get a typesafe Chooser class without doing anything else that is complicated?
- What would Liskov have to say about this class? How should it evolve to address her concerns? What is the appropriate place to deal with the problem? Does an invariant help? Is that a rep-invariant, or some other kind of invariant?

2.18 In class 8C

```
public class BoundedQueue {

    protected Object rep[];
    protected int front = 0;
    protected int back = -1;
    protected int size = 0;
    protected int count = 0;

    public BoundedQueue(int size) {
        if (size > 0) {
            this.size = size;
            rep = new Object[size];
            back = size - 1;
        }  }

    public boolean isEmpty() { return (count == 0); }

    public boolean isFull() { return (count == size); }

    public int getCount() { return count; }

    public void put(Object e) {
        if (e != null && !isFull()) {
            back++;
            if (back >= size)
                back = 0;
            rep[back] = e;
```

```
            count++;
        }  }

    public Object get() {
        Object result = null;
        if (!isEmpty()) {
            result = rep[front];
            rep[front] = null;
            front++;
            if (front >= size)
                front = 0;
            count--;
        }
        return result;
    }
}
```

Generify!

- Can you add a putAll() method? A getAll() method?
- Recall that we used this same example in in-class 6 as a vehicle for applying Liskov's ideas to make code easier to understand.

2.19 In class 9A

Consider Bloch's `Point/ColorPoint` example. For today, ignore the `hashCode()` issue.

```
public class Point {  // routine code
    private int x; private int y;
    ...
        @Override public boolean equals(Object obj) {  // Standard recipe
        if (!(obj instanceof Point)) return false;

        Point p = (Point) obj;
        return p.x == x && p.y == y;
    }
}
```

```
public class ColorPoint extends Point {  // First attempt: Standard recipe
    private COLOR color;
    ...
        @Override public boolean equals(Object obj) {
        if (!(obj instanceof ColorPoint)) return false;

        ColorPoint cp = (ColorPoint) obj;
        return super.equals(obj) && cp.color == color;
    }
}

public class ColorPoint extends Point {  // Second attempt: DON'T DO THIS!
    private COLOR color;
    ...
        @Override public boolean equals(Object obj) {
        if (!(o instance of Point)) return false;

        // If obj is a normal Point, be colorblind
        if (!(obj instanceof ColorPoint)) return obj.equals(this);

        ColorPoint cp = (ColorPoint) obj;
        return super.equals(obj) && cp.color == color;
    }
}
```

1. What is the `equals()` contract? What is the standard recipe?
2. Why does Bloch use the `instanceof` operator in the standard recipe?
3. Write client code that shows a contract problem with the first attempt at `ColorPoint`.
4. Write client code that shows a contract problem with the second attempt at `ColorPoint`.
5. Some authors recommend solving this problem by using a different standard recipe for `equals()`.
    - What's the key difference?
    - Which approach do you want in the following code:

```
public class CounterPoint extends Point
                                    private static final AtomicInteger counter =
                                    new AtomicInteger();

public CounterPoint(int x, int y) {
    super (x, y);
    counter.incrementAndGet();
}
public int numberCreated() { return counter.get(); }

@Override public boolean equals (Object obj) { ??? }
}


// Client code:

Point p = PointFactory.getPoint();   // either a Point or a CounterPoint
Set<Point> importantPoints =   // a set of important points
    boolean b = PointUtilities.isImportant(p);  // value?
```

2.20 In class 9B

Consider a variation of Liskov's `IntSet` example (Figure 5.10, page 97)

```
public class IntSet implements Cloneable {
    private List<Integer> els;
    public IntSet () { els = new ArrayList<Integer>(); }
    ...
        @Override
        public boolean equals(Object obj) {
        if (!(obj instanceof IntSet)) return false;

        IntSet s = (IntSet) obj;
        return ???
            }

    @Override
    public int hashCode() {
```

```
        // see below
    }

    // adding a private constructor
    private IntSet (List<Integer> list) { els = list; }

    @Override
    public IntSet clone() {
        return new IntSet ( new ArrayList<Integer>(els));
    }

}
```

1. How should the `equals()` method be completed?
2. Analyze the following ways to implement `hashCode()`? If there is a problem, give a test case that shows the problem.
   (a) not overridden at all
   (b) return 42;
   (c) return `els.hashCode()`;
   (d) `int sum = 0; for (Integer i :  els) sum += i.hashCode(); return sum;`
3. What's the problem with `clone()` here? Give a test case that shows the problem.
4. Fix `clone()` in two very different ways.

2.21 In class 10A

Consider Bloch's `InstrumentedHashSet`, `InstrumentedSet`, and `ForwardingSet` examples:

```
public class InstrumentedHashSet<E> extends HashSet<E>{
    private int addCount = 0;
    public InstrumentedHashSet() {}

    @Override public boolean add(E e){
        addCount++;
        return super.add(e);
```

```
    }
    @Override public boolean addAll(Collection<? extends E> c){
        // What to do with addCount?
        return super.addAll(c);
    }
    public int getAddCount(){ return addCount; }
}
public class InstrumentedSet<E> extends ForwardingSet<E>{
    private int addCount = 0;

    public InstrumentedSet(Set<E> s){ super(s); }
    @Override public boolean add(E e){ addCount++; return super.add(e); }
    public int getAddCount(){ return addCount; }
}
public class ForwardingSet<E> implements Set<E> {
    private final Set<E> s;

    public ForwardingSet(Set<E> s){ this.s = s; }
    public           boolean add(E e)        { return s.add(e);     }
    public           boolean remove(Object o){ return s.remove(o);  }
    @Override public boolean equals(Object o){ return s.equals(o);  }
    @Override public int     hashCode()      { return s.hashCode(); }
    @Override public String  toString()      { return s.toString(); }
    // Other forwarded methods from Set interface omitted
}
```

Consider also the following client code:

```
Set<String> r = new HashSet<String>();
r.add("ant"); r.add("bee");

Set<String> sh = new InstrumentedHashSet<String>();
sh.addAll(r);

Set<String> s =  new InstrumentedSet<String>(r);
s.add("ant"); s.add("cat");

Set<String> t = new InstrumentedSet<String>(s);
t.add("dog");
```

```
r.remove("bee");
s.remove("ant");
```

1. How do you think the `addCount` variable should be updated in the `addAll()` method in `InstrumentedHashSet`?
   (a) Why is this a hard question?
   (b) What does the answer say about inheritance?
   (c) Does `equals()` behave correctly in `InstrumentedHashSet?`
2. Given your previous answer, what is the value of `sh.addCount` at the end of the computation?
3. Consider the `InstrumentedSet` solution. Besides being correct (always a plus!) why is it more general than the `InstrumentedHashSet` solution?
4. At the end of the computation, what are the values of: `r`, `s`, and `t`?
5. What would a call to `s.getAddCount()` return at the end of the computation?
6. At the end of the computation, what are the values of: `r.equals(s)`, `s.equals(t)`, and `t.equals(s)`?
   - Are there any problems with the `equals()` contract?
7. Would this still work if you globally replaced sets with lists?
8. Would this still work if you globally replaced sets with collections?

   Note: There is a lot going on in this example. I highly recommend that you play with the code until you understand it.

2.22 In class 10B

```
public class Super {
    public Super() {
        overrideMe();
    }

    public void overrideMe () {
    }
```

```
}
public final class Sub extends Super {

    private final Date date;  // filled in by constructor

    public Sub() {
        date = new Date();
    }
    @Override public void overrideMe () {
        System.out.println(date);
    }

    public static void main (String[] args) {
        Sub sub = new Sub();
        sub.overrideMe();
    }
}
```

1. What is the pattern, and how common is it?
2. What does the main method do, and why?
3. Which of Bloch's rules does this example break?
4. What does this example mean for `Cloneable` interface and the `clone()` method?
5. What does this example mean for `Serializable` interface and the `readObject()` method?
6. To what extent does this rule generalize to producer methods?

2.23 In class 10C

Consider a mutable complex number class:

```
public class MComplex {
    double re; protected double im;

    public MComplex (double re, double im) { this.re = re; this.im = im; }

    public double getReal()      { return re; }
```

```
public double getImaginary() { return im; }

public void setReal(double re)      { this.re = re; }
public void setImaginary(double im) { this.im = im; }

public void add (MComplex c) { re += c.re; im += c.im; }

public void subtract (MComplex c) { re -= c.re; im -= c.im; }

public void multiply (MComplex c) {
    double r = re * c.re - im * c.im;
    double i = re * c.im + im * c.re;
    re = r; im = i;
}

public void divide (MComplex c) {
    double den = c.re * c.re + c.im * c.im;
    double r = (re * c.re - im * c.im) / den;
    double i = (re * c.im + im * c.re) / den;
    re = r; im = i;
}

@Override public boolean equals (Object o) {
    if (o == this)               return true;
    if (!(o instanceof MComplex)) return false;
    MComplex c = (MComplex) o;

    // See Bloch page 43 to find out why to use compare() instead of ==
    return Double.compare(re, c.re) == 0 &&
        Double.compare(im, c.im) == 0;
}

@Override public int hashCode () {
    int result = 17 + hashDouble(re);
    result = 31 * result + hashDouble(im);
    return result;
}

private int hashDouble (double val) {
    long longBits = Double.doubleToLongBits(val);
```

```
        return (int) (longBits ^ (longBits >>>32));
    }

    @Override public String toString() { return "(" + re + " + " + im + "i)"; }
}
```

Before we get to immutability, consider the method contracts. Where do the various contracts "come from", and is there anything in the (missing) JavaDoc that might require a bit of research?

Apply each of Bloch's 5 rules for making a class immutable:

1. Don't provide any methods that modify the object's state. How do you handle the mutators?
2. Ensure that no methods can be overridden.
   - Why is this a problem? Show me!
   - Fix the problem:
     - Change the class declaration, or
     - Change the method declarations, or
     - Change the constructor visibility.
3. Make all fields final.
4. Make all fields private.
   - Is there a significant difference in visibility between re and im?
5. Ensure exclusive access to any mutable components.

2.24 In class 11

This is a JUnit theory exercise.

1. Write a JUnit theory that captures the symmetry property of the `equals()` method.
2. Create `@DataPoints` from Bloch's `Point`, `ColorPoint` classes. So that we're all on the same page, create 1 `null` reference, 1 `Point` object and 2 `ColorPoint` objects.
3. Given this set of data points:

- How many combinations are considered by the theory?
- How many combinations make it past the preconditions of the theory?
- How many combinations make it to the postcondition of the theory?

4. What happens to this theory and the accompanying data points when favoring composition over inheritance?

5. Repeat the exercise for the transitive property for `equals()`.

6. Recall the `equals()` and `hashCode()` discussion in Bloch. Write a JUnit theory that encodes the consistency property between `equals()` and `hashCode()`.

7. Build a toy example that violates the theory. Fix the toy example so that the theory is no longer violated.

8. Consider the `Comparable` interface: what properties should be checked with theories?

2.25 In class 12A

Consider the following (bad) Java, implementing the "C style" enum pattern:

```
public class Coins {
    public static final int PENNY = 1;
    public static final int NICKLE = 5;
    public static final int DIME = 10;
    public static final int QUARTER = 25;
}
```

1. Give example code that illustrates a type safety problem with `Coins`. Work through a range of expressions from "probably ok" to "clearly wrong".

2. What code would you need to turn a nickel into a string? Explain how this could go wrong at runtime.

3. What code would you need to iterate through the coins?

4. Would extensions to this particular enum be likely to require recompilation of client code? Explain.

5. Write a decent Java Enum for coins.

6. Turn a nickle into a string.

7. Iterate though the coins.

Consider Bloch's example:

```
// Abuse of ordinal to derive an associated value - DON'T DO THIS
public enum Ensemble {
    SOLO,   DUET,   TRIO,  QUARTET, QUINTET,
    SEXTET, SEPTET, OCTET, NONET,   DECTET;

    public int numberOfMusicians() { return ordinal() + 1; }
}
```

Explain why it's wrong, fix it, and add another enum with an overlapping number of musicians.

2.26 In class 12B

This is a recap exercise based on the map-based implementation of Liskov's polynomial example: MapPoly

1. How are the following polynomials represented?

   - 0
   - $3 - 7x^4$

2. Bloch would not accept that the MapPoly class is immutable. Why not? Show how it would be possible to provide mutable behavior with the class if Bloch's problem isn't fixed. Fix the problem, and implement any other changes Bloch suggests, even if they don't compromise immutability in this particular example.

3. Write a reasonable rep-invariant for `MapPoly`. How would this rep-invariant change if the zero `Poly` had an alternate representation.

4. Provide reasonable implementations of `equals()` and `hashCode()`. Explain why you believe your implemetations are appropriate.

5. As written, the contract for the `coeff()` method is inconsistent with other contracts in the class.

   - What is the inconsistency with the contract?
   - Fix the inconsistency with the contract.
   - Fix the code to match the revised contract.

6. Argue that the implementation of the `coeff()` method is correct (with respect to your repaired contract, of course.)

7. Consider implementing `Cloneable` for this class. Decide whether Bloch would think this is a good idea and provide justification for your answer. Note: You don't have to actually implement anything for this question.

8. See if you can come up with a theory about Polys and implement it in JUnit. (Polys are math objects, so there should be theories!) Here's a suggestion: Think about the relationship between the degrees of two Polys being multiplied and the resulting degree.

2.27 In class 13

How well are you prepared for the final? This exercise should help you find out. Piazza discussions encouraged!

```
public class Stack {
    private Object[] elements; private int size = 0;

    public Stack() { this.elements = new Object[0]; }

    public void push (Object e) {
        if (e == null) throw new NullPointerException("Stack.push");
        ensureCapacity(); elements[size++] = e;
    }

    public void pushAll (Object[] collection) { for (Object obj: collection) { push(obj

    public Object pop () {
        if (size == 0) throw new IllegalStateException("Stack.pop");
        Object result = elements[--size];
```

```
        // elements[size] = null;
        return result;
    }

    @Override public String toString() {
        String result = "size = " + size;
        result += "; elements = [";
        for (int i = 0; i < elements.length; i++) {
            if (i < elements.length-1)
                result = result + elements[i] + ", ";
            else
                result = result + elements[i];
        }
        return result + "]";
    }
}
```

1. Write a contract for `push(Object e)`.
2. What is wrong with `toString()`? Fix it.
3. What rep-invariant is likely broken? Fix it. This includes writing a suitable rep-invariant.
4. How would Bloch's Item 25: *Prefer Lists to Arrays* apply here? Would it make the rep-invariant simpler?
5. How would you argue that that `pop()` is correct (or not)?
6. As `Stack` is written, `pushAll()` requires special documention? Why? What would Bloch suggest as an alternative?
7. Override `equals()`. What else do you have to do? Do that too.
8. Generify. What should happen to the parameter for `pushAll()`? Why?
9. Suppose we decide to implement the `Cloneable()` interface. In what ways would Bloch think we would likely get it wrong? What would Bloch recommend instead?

3 HW Assignments

3.1 Assignment 1

3.1.1 Goal

- Getting started on Piazza.
- Getting your group together.

There are two parts to this assignment:

- Post a brief intro about yourself on the course Piazza page. For any credit, the posting must:
  - be a follow-up to my introduction. In other words, all intros need to be in the same thread.
  - Include a photo appropriate in size, content, and orientation.
- Your **group** should communicate the composition of your group to me (and the GTA) on Piazza. If you group is sticking with the random assignment, just confirm that. If you have a new group, tell us the composition, and we'll edit the post to reflect the change.

3.1.2 Grading Criteria

- Your individual Piazza post adhers to my instructions. (That is, no sideways pictures, no oversize pictures, etc.)
- You are in a group.

3.2 Assignment 2

3.2.1 Goals: Contracts

For the second assignment, you'll build a *very* small piece of Java for a contract with preconditions, transform the contract so that all preconditions become postconditions, and then re-implement appropriately.

- Consider a method that calculates the number of months needed to pay off a loan of a given size at a fixed *annual* interest rate and a fixed *monthly* payment. For instance, a $100,000 loan at an 8% annual rate would take 166 months to discharge at a monthly payment of $1,000, and 141 months to discharge at a monthly payment of $1,100. (In both

of these cases, the final payment is smaller than the others; I rounded 165.34 up to 166 and 140.20 up to 141.) Continuing the example, the loan would never be paid off at a monthly payment of $100, since the principal would grow rather than shrink.

Define a Java class called `Loan`. In that class, write a method that satisfies the following specification:

```
/*
@param principal:  Amount of the initial principal
@param rate:       Annual interest rate  (8% rate expressed as rate = 0.08)
@param payment:    Amount of the monthly payment
*/
public static int months (int principal, double rate, int payment)
    // Requires: principal, rate, and payment all positive and payment is sufficiently
    // Effects:  return the number of months required to pay off the principal
```

Note that the precondition is quite strong, which makes implementing the method easy. You should use double precision arithmetic internally, but the final result is an integer, not a floating point value. The key step in your calculation is to change the principal on each iteration with the following formula (which amounts to monthly compounding):

```
newPrincipal = oldPrincipal * (1 + monthlyInterestRate) - payment;
```

The variable names here are explanatory, not required. You may want to use different variables, which is fine.

To make sure you understand the point about preconditions, your code is required to be minimal. Specifically, if it possible to delete parts of your implementation and still have it satisfy the requirements, you'll earn less than full credit.

- Now modify `months` so that it handles **all** of its preconditions with exceptions. Use the standard exceptions recommended by Bloch. Document this with a revised contract. You can use JavaDoc or you can simply identify the postconditions.

3.2.2 Grading Criteria

- Adherence to instructions.
- Minimal implementation.

- Preconditions are correctly converted to exceptions.
- Syntax: Java compiles and runs.

3.3 Assignment 3

3.3.1 Goals: Data Abstraction / Mutability

Rewrite MapPoly, my map-based version Liskov's Poly so that it is *mutable*. Keep the same representation.

Rewrite the overview, the method signatures, the method specifications, and the methods themselves. You do not need to rewrite the abstraction function and representation invariant for this exercise.

Turn in a **story**. This means that it is possible to grade your assignment simply by reading it, as if it were part of a textbook. In particular, every place you make a decision to change something in the code (or not), you should have a description of what you did (or didn't do) and why you did (or didn't do) it.

Remember that part of your group is responsible for synthesizing a solution, and part of your group is responsible for checking the result.

3.3.2 Grading Criteria

- Correct transformation of Poly
- Clarity of your story.
- Reasonable division of synthesis vs. checking.

3.4 Assignment 4

3.4.1 Goals: Rep-Invariants, contracts, tests

Revisit the mutable Poly example from assignment 3. That is, use the one based on a map, not an array.

1. Implement `repOk()`.
2. Introduce a fault (i.e. "bug") that breaks the rep-invariant. Try to do this with a small (conceptual) change to the code. Show that the rep-invariant is broken with a JUnit test.
3. Analyzed your bug with respect to the various contracts/methods in Poly. Are all/some/none of the contracts violated?

4. Do you think your fault is realistic? Why or why not?

As in assignment 3, your deliverable is a **story**, with exactly the same rationale. Take screenshots (e.g. of failing JUnit tests) as necessary to make your case.

3.4.2 Grading Criteria

- Correctness of solution
- Clarity of story

Note: If your group had trouble with the previous assignment, feel free to appeal to your classmates to post a sample solution on Piazza.

3.5 Assignment 4-1

3.5.1 Goals: Understanding Program Verification through Hoare Logic

Do the in-class exercise with your group and submit it on BB. More specifically, you will do the below two tasks:

1. Prove the program using the following the loop invariant: `i <= N`.
   (a) Clearly reason why this is a loop invariant
   (b) Compute the weakest precondition `wp` of the program wrt the post conditiong `Q`
   (c) Compute the verification condition `vc (P => wp(..))`, and
   (d) Analyze the `vc` to dertermine whether the program is proved or not
2. Repeat the above task a different loop invariant: `N >= 0`

3.5.2 Grading Criteria

- Correctness of solution

Note: If your group had trouble with the assignment, feel free to appeal to your classmates to post a sample solution on Piazza.

3.6 Assignment 5

3.6.1 Goals: Immutablity via Bloch Item 50

Revisit the Period example.

Implement a satisfying solution to question 3. That is, you should not only break the immutability of the `Period` class by writing a suitable sublcass, but you should also develop a plausible case where a client ends up "in trouble" due to the loss of immutability.

Turn in a **story**.

3.6.2 Grading Criteria

Grading is in part the technical aspect of breaking immutability, and in part that your client case is plausible.

3.7 Assignment 6

3.7.1 Goals: Type Abstraction

Consider the following `Market` class.

```
class Market {
    private Set<Item> wanted;           // items for which prices are of interest
    private Bag<Item, Money> offers;    // offers to sell items at specific prices
    // Note:  Bag isn't a Java data type.  Here, the bag entries are pairs.

    public void offer (Item item, Money price)
    // Requires: item is an element of wanted
    // Effects:  add (item, price) to offers

        public Money buy(Item item)
    // Requires: item is an element of the domain of offers
    // Effects: choose and remove some (arbitrary) pair (item, price) from
    //          offers and return the chosen price
        }
```

1. Suppose that offers are only accepted if they are lower than previous offers.

```
class Low_Bid_Market extends Market {
    public void offer (Item item, Money price)
    // Requires: item is an element of wanted
    // Effects:  if (item, price) is not cheaper than any existing pair
    //           (item, existing_price) in offers do nothing
    //           else add (item, price) to offers
```

Is `Low_Bid_Market` a valid subtype of `Market`? Appeal to the methods rule to back up your answer.

2. Suppose that the `buy()` method always chooses the lowest price on an item.

```
class Low_Offer_Market extends Market {
    public Money buy(Item item)
    // Requires: item is an element the domain of offers
    // Effects: choose and remove pair (item, price) with the
    //          lowest price from offers and return the chosen price
```

Is `Low_Offer_Market` a valid subtype of `Market`? Appeal to the methods rule to back up your answer.

3.7.2 Grading Criteria

This is purely a "paper and pencil" exercise. No code is required. Write your answer so that it is easily understandable by someone with only a passing knowledge of Liskov's rules for subtypes.

3.8 Assignment 7

3.8.1 Goals: Polymorphic Abstraction.

A `Comparator` based on absolute values is problematic. Code up the comparator and then write client code that illustrates the problem. Use a *lambda function* to implement the comparator. Explain what is wrong in a brief summary statement. Your explanation of the problem must be phrased in terms of a violation of the contract for `Comparator`.

To emphasize that this contract problem is real, your code should create two Java sets, one a `HashSet`, and the other a `TreeSet`. The `TreeSet` should order items with your absolute value comparator. Your example should add the same integers to both sets, yet still end up with sets that are different. Your summary statement should explain why.

3.8.2 Grading Criteria

As for other recent assignments, your deliverable is a clear, concise story that demonstrates completion of the assignment.

3.9 Assignment 8

3.9.1 Goals: Generics

Consider the BoundedQueue example from in-class exercise #8C.

Complete the generic part of the exercise: The result should be fully generic, and there should not be any compiler warnings. You should adopt Bloch's advice about lists vs. arrays; doing so will eliminate the need for many of the instance variables.

Keep the same methods, but update the behavior (and document with contracts!) to include exception handling for all cases not on the happy path.

Include the constructor in your considerations. In particular, consider whether you think a zero-sized buffer is a reasonable possibility. Document your reasoning. This is less about a right vs. wrong answer than a careful consideration of the consequences of the decision.

Add `putAll()` and `getAll()`. Define the method signatures carefully. Use exception-handling consistent with that for `get()` and `put()`. Use bounded wildcards as appropriate. Note that `putAll()` has a special case where there isn't sufficient space in the bounded queue. Adopt a solution you think Bloch and/or Liskov would approve of. In particular, Bloch prefers that when methods throw exceptions, there is no change to the state of the object.

3.9.2 Grading Criteria

As before, turn in a clear, concise story demonstrating completion of the assignment.

3.10 Assignment 9

3.10.1 Goals: `Object` class contracts.

As it happens, Liskov's implementation of `clone()` for the `IntSet` class (see figure 5.10, page 97) is wrong.

1. Use the version of `IntSet` from the in-class exercise. Implement a subtype of `IntSet` to demonstrate the problem. Your solution should include appropiate executable code in the form of JUnit tests.

2. Provide a correct implementation of `clone()` for `IntSet`. Again, give appropriate JUnit tests.

3. Correctly override `hashCode()` and `equals()`. As discussed in the class exercise, the standard recipe is not appropriate in this (unusual) case.

3.10.2 Grading Criteria

In addititon to code and tests, your deliverable is a story. Explain what is going on at each stage of the exercise. The GTA will primarily grade your story.

3.11 Assignment 10

3.11.1 Goals: Favoring composition over inheritance. Bloch, Item 18.

Consider the `InstrumentedSet` example from Bloch Item 18 (as well as in-class exercise #10A).

1. Replace `Set` with `List`. There is no problem with `equals()`. Why not?

2. Replace `Set` with `Collection`. Now `equals()` does not satisfy its contract.

   - Explain why there is a problem.
   - Demonstrate the problem with a suitable JUnit test.

3.11.2 Grading Criteria

The GTA will look for correct responses, appropriate JUnit tests, and plausible explanations when doing the grading.

3.12 Assignment 11

3.12.1 Goals: Applying lessons learned.

You have a choice of possible assignments:

1. Consider one of the `copyOf()` methods in the Java Arrays utility class. Bloch uses this method in his `Stack` example. Code a corresponding method in C++, changing the argument list as necessary. Provide a

specification for the C++ code by translating the JavaDoc and adding preconditions as necessary. Explain what this exercise demonstrates about C++ type safety.

2. For most of the semester, we have focused on design considerations for constructing software that does something we want it to do. For this last assignment, I would like students to appreciate just how vulnerable software is to malicious parties intent on attacking their software.

   There are two attacks documented in Bloch's Item 88: *Write* `readObject()` *methods defensively.* One is called `BogusPeriod`, and the other is called `MutablePeriod`. Implement either (your choice) of these attacks (basically involves typing in code from Bloch) and verify that the attack takes place.

3. A different source of security vulnerabilities in Java also involve serialization. Bloch (and others) recommend "cross-platform structured data representations" (e.g. JSON or Protocol Buffers) as safe alternatives. Develop a simple serialization example in Java and convert it into a safe alternative (probably, JSON is easier to use, since it is text-based). To make the example more interesting, use some objects types that are not directly supported.

4. Find some existing (Java) code that uses the "int enum pattern" and refactor it to use Java `Enums` instead. Identify any type-safety issue you uncover in the existing code. To make the exercise interesting, extend your enums beyond simple named-constants in one of the ways discussed by Bloch in Item 34.

5. Where appropriate, code up, as JUnit theories, constraints for classes that implement the Java `Comparable` interface. Note that there is significant overlap with the in-class exercise. Note also that the Comparable interface is generic; hence, you should use generics in your JUnit test class.

6. Gain experience with one of the property-based testing tools. I suggest a Java-based one (such as jqwik). One way to do this is work through one of the articles linked on the jqwik site.

3.12.2 Grading Criteria

In each case, the deliverable is a story. Write a brief report, and include enough evidence (output, screen shots, etc.) that the GTA can figure out

that you actually completed the assignment.

4 Quiz Guides

Note: it's possible that your quiz involves last week's topic. Be prepared for both!

4.1 Guide 1

Quiz 1 will revisit the example from In-Class Exercise 0. I'll ask you about the **first** of the two given `equals()` methods, as well as "corner" cases where this method might do something odd.

This won't be a deep-dive; that comes later. But you should be able to identify specific inputs that lead to corner case behavior. You should be able to assess code behavior on specific inputs.

Quiz 1 may also include items from the syllabus and from the readings. Please read both carefully!

4.2 Guide 2

Quiz 2 will focus on Liskov, Chapters 3-4 and Bloch 10. Specifically, you should be able to explain the code and the contracts for in-Class exercise 1A. As part of this, you should be able to transform preconditions into postconditions via the exception handling mechanism, and you should be able to incorporate Bloch's advice on exceptions into this transformation.

4.3 Guide 3

Quiz 3 will focus on the first part of Liskov 5. You should be able to manipulate the IntSet and Poly examples. You should understand basic mutability - that is, the specification of mutators in mutable classes and producers in immutable classes. You should be able to convert the specification of a simple mutable class to an immutable one, and vice versa.

4.4 Guide 4-1

Quiz 4-1 will focus on program verification using Hoare tripple. You should understand and able to do examples we have discussed in class. In particular, I'd suggest modifying the examples or specifications or invariants and see if the verification process still works or fails.

4.5 Guide 4

Quiz 4 will focus on abstraction functions, rep-invariants, and verification. You should understand, evaluate, and modify the abstraction functions and rep-invariants for simple variations on examples we have discussed in class. You should also understand the verification of methods with respect to their specifications. If I give you a specification, and a Java implementation, you should be able to analyze (informally) whether the method is correct. In particular, I'd suggest studying the verification of the Members example, which we covered in the in-class exercise.

4.6 Guide 5

Iteration abstraction is the focus of Quiz 5. You should understand the abstraction functions for iterators, as well as the examples Liskov covers.

Also on the agenda is Bloch 3rd edition, Chapter 8 (Methods).

4.7 Guide 6

Type abstraction is the focus of Quiz 6. In addition to the basic Java mechanisms for implementing type abstraction, you should understand section 7.9, particularly the "signature" rule, the role of preconditions and postconditions in the "methods" rule, and simple applications of the "properties" rule. You should be prepared to analyze example specifications for overridden methods.

4.8 Guide 7

Two possible foci for Quiz 7: Java's lambda expressions as explored in the in-class exercise. The element subtype vs. related subtype approaches to polymorphism and how they are implemented in Comparable vs. Comparator.

4.9 Guide 8

Quiz 8 will focus both Liskov's treatment of polymorphism and Bloch's treatment of lambda expressions.

To make this concrete, we'll focus on the `Comparator` interface. You should be prepared to evaluate various implementations of this interface against the contract for the interface, with the ability to explain why certain violations of the contract could lead to trouble (e.g. when used in a collection

framework such as `TreeSet`). You should also be prepared to manipulate this interface via lambda expressions (e.g. when used in a collection framework such as `TreeSet`).

This homework should be excellent preparation.

4.10 Guide 9

Quiz 9 will focus on Bloch's Chooser example. There is a lot going on in this example. Not only does it illustrate many of the points Bloch makes about generics, but it is also a good place to apply what we learned in Liskov about analyzing data types. Note that `Chooser` is very similar to Liskov's `IntSet` class.

4.11 Guide 10

Quiz 10 will focus on the Bloch's treatment of `Object` class methods.

In particular, you should be able to identify defective implementations of `equals()`, `hashCode()`, and `clone()`, explain what's wrong, and repair appropriately. The assessments will be based on the examples we study in class.

4.12 Guide 11

Quiz 11 will focus on the Bloch Chapter 4 with special emphasis on Item 17: Minimize mutability and Item 18: Favor composition over inheritance. In particular, you should be prepared to apply Bloch's rules for making a class immutable to a simple example and you should understand the various aspects of Bloch's InstrumentedSet example (code on page 90).

4.13 Guide 12

Quiz 12 will focus on the contract model in JUnit theories. The specific examples will be variations from In-Class 11.

5 Reflection

For each of the following, answer these two questions first:

1. List the names of students in your group.
2. Did everyone in your group contribute to the discussion of your solutions to this reading quiz? If not, who did not?

5.1 Reflection 1

1. Much of the material explores the connection between preconditions and exception handling. Were there any aspects of this connection that surprised or confused anyone in your group? If so, explain. If not, where did you learn this material?

2. Liskov and Bloch have different advice with respect to checked vs. unchecked exceptions. Which approach do you find more persuasive, and why?

3. Preconditions are often characterized as "bad" from a security perspective. If you think you know why this is, please explain. If you are unsure, say so and try to explain why the you find the connection between preconditions and security confusing.

5.2 Reflection 2

1. If you sat down to design a new class, would the result likely be mutable or immutable? Why?

2. In her presentation, Liskov doesn't cover all the requirements for immutability. (In fairness, these requirements weren't well understood at the time she wrote her text.) Do you know what she's missing and why it's important? If so, briefly explain. (We'll cover those requirements later in the semester.)

3. Based on your experience, what do you think the major advantage is of immutability over mutability? mutability over immutability?

5.3 Reflection 3

1. Have you ever explicitly considered invariants when deciding how to implement a Java class? If so, can you give an example?

2. Please explain what you think it means to to correctly override the toString() method. Base your answer on your understanding **before** enrolling in SWE 619.

3. How do you decide whether you have implemented a Java method correctly? Again, base your answer on your understanding **before** enrolling in SWE 619.

5.4 Reflection 4

1. Iteration is a basic concept, yet Liskov devotes an entire chapter to it. What, if anything, did you find in Liskov's presentation of iteration abstraction that is new to you?

2. Bloch's `Period` class (Item 50) has a lot going on in it. We'll revisit the this example in an in-class exercise. What, if anything, did you find confusing in this example?

5.5 Reflection 5

1. Liskov 7 develops rules for assessing the correctness of subtypes. What do you think the connection is between these rules and the rules for verification addressed in Chapter 5?

1. Consider the Java Set interface and two subtypes: HashSet and TreeSet. Do you think the abstract state for these three interfaces/classes are identical or different? (You might want to spend some time in the JavaDoc before jumping to a conclusion; there is a specific answer in there!)

5.6 Reflection 6

1. Can you explain why Java has both a Comparable interface and a Comparator interface?

2. How familiar is your group with the Java "anonymous class" and "lambda" constructs?

3. Can you explain the connection between anonymous classes and lambda expressions?

5.7 Reflection 7

1. Can you explain the basic role of generics in the Java language?

2. Do you have experience generifying Java classes? Explain.

3. Bloch explains how bounded wildcards can address certain limitations in the use of generics in inheritance settings. If you can, give a brief description of how this works. (If not, that's fine; we'll address in class.)

5.8 Reflection 8

1. Have you overridden the equals() or the hashCode() methods? In light of Bloch's discussion of both methods, do you think your implementations were correct?

2. Have you overridden the clone() method? Do you understand why inheritance is a particular concern for overridding this method?

3. What similarities and differences do you see between how Liskov and Bloch treat the toString() method?

5.9 Reflection 9

1. What is your comfort level with "regular" logic? What aspect of using logic do you find most challenging?

2. Do you have any experience learning and/or using temporal logic?

3. How familiar/comfortable are you with finite state machines?

5.10 Reflection 10

1. Bloch discusses specific rules for making a class immutable. Did you find any of these rules confusing?

2. Bloch's InstrumentedHashSet example demonstrates how inheritance can break encapsulation. Does the JavaDoc for HashSet, Set and/or Collection follow the Bloch's Item 19 advice for documenting for inheritances?

3. Bloch's InstrumentedSet example has a lot going on in it. What aspects, if any, of this example did you find confusing?

5.11 Reflection 11

1. How would you rate your experience with writing (ordinary) tests in the JUnit framework? Use a scale from "A few times for class" to "I do that professionally".

2. JUnit theories are the JUnit implementation of "property-based" testing. Have you every written a property-based test?

3. JUnit theories are included on the syllabus because they show how the precondition/postcondition model applies beyond method contracts. Does the pre/post model for JUnit theories make sense to you?

5.12 Reflection 12

1. Is there anything about property based testing that you still find confusing?
2. Have you ever used a "C style" enum? If so, at the time, did this seem reasonable or ridiculous?
3. This week's in-class exercise is a recap. Is there a topic (or two) we've covered that you think you need more practice with?

6 Files

- LiskovSet.java
- Poly.java

7 Links

- Syllabus
- Schedule