Topic 4: Guidelines for Class Design

Part 2: Encapsulation Ch3.4

Encapsulation

Encapsulation

- Motivation:
- Consider an implementation of the Day class that uses public instance variables

```
public class Day{
    public int year;
    public int month;
    public int date;
    ...
}
```

- Any change in the internal implementation of the class would affect the clients (the users) of that class.
- As a general rule, only expose enough functionality to do the job.

Encapsulation

- **Encapsulation** is the bundling of related data and operations on that data (into a class) in order to restrict client access to specific parts of that class.
- Breaking encapsulation is generally a bad idea because it inhibits change.
- Hidden components can change easily
  - Is a technique to “future proof” your code
- Benefits of Encapsulation:
  - Reduces the scope of change
  - Reduces developer’s cognitive load

Accessors and Mutators

- **Mutator methods:** change the state of an object
- **Accessor methods:** reads the state of an object
- **Immutable**: an object with no methods that change its visible state
  - Once created, you cannot change it's (visible) state.
- Q: Is DayThree immutable?
  - Lazy conversion changes its private fields.
  - externally it has the same state.
- Immutability implications for Day
  - `addDays()` must return a new Day object
- Similar to `String.toLower()`:

```
      String msg = "Hello World".toLower();
```

Why Immutable?

- Automatically adding mutators for every instance property may lead to unwanted results (Dating.java)
- Shared Reference
  - Object references of immutable objects can be freely shared without the worry of tampering
- Thread safe
- As a general rule, make your class immutable whenever possible

Shared Reference Problem

- Client with mutable Date object
  - Date is mutable (i.e. `setTime()`)
  - (Person.java) (SharedReference.java)
  - What is the problem? How did this problem occur?

- To protect a class (i.e. Person) from unexpected changes
  - Use an immutable object
  - Use a clone to return a duplicate object

Accessor “Safety”

- Is it "safe" (i.e., unchangable) for an object's accessor to return
  - a reference to a field of a mutable type? (Ex: `Date`)
  - a reference to a field of a immutable type? (Ex: `String`)
  - a primitive typed field? (Ex: `int`)
- Immutable objects prevent (unexpected) change.
- Only make an object mutable if you expect it to change over time
  - Ex: A message queue, a person, etc.

`final` Fields

- A field can be marked `final` meaning that the variable cannot be made to reference another object
  (or change its value if a primitive).
- Can be assigned a value either:
- a)

```
private class Car {
        final private String MODEL = "X";
}
```

- b)

```
private class Car {
        final private String MAKE;
        public Car() {
                  MAKE = "Tesla";
        }
}
```

Quick `final` Example

Which lines contain errors?

```
public class Final {
    public final int MAX_PERCENT = 100;
    private final ArrayList<Person> list;
    public Final() { list = new ArrayList<Person>(); }

    // ...
    public void doSomething() {
        // a) Constant to variable & change?
        int w = MAX_PERCENT;
        w++;

        // b) Change constant?
        MAX_PERCENT = 50;

        // c) Change which object?
        list = new ArrayList<Person>();

        // d) Access from object?
        int x = list.size();
        x++;

        // e) Change object's state?
        list.add(new Person("Bobby", 25, new Date()));
    }
}
```

Command/Query Separation

Guidelines

Command-Query Separation

- **Command**: A method which changes an objects
  Query: A method which returns the state of an object without changing it.
- Command-Query Separation Guideline - Each method should do at most one of:
  - Change state of an object.
  - Return a value/part of the state.
- Recall: an object with no command methods is called **immutable**

Command-Query Separation (2)

Violation

- Example violation of Command-Query Separation

```
public class BankAccount {
        private int balance = 0;

        public int getBalance(int value) {
                return balance -= value;
        }
}
```

- Solution: Separate into two methods:

Iterator

- Iterators – allow you to iterate over a collection in java.
  - Used in the collections to retrieve objects one by one
  - Iterators allow the caller to remove elements from the underlying collection during the iteration
- Iterator is an interface that defines three methods:

`public boolean hasNext();`	Returns true if the iteration has more elements
`public Object next();`	Returns the next element in the iteration. If no more elements, Throws **NoSuchElementException**
`public void remove();`	Removes the next element in the iteration

Iterator: Example

- Complete this function, using an iterator, to add up all numbers in the following collection: (IteratorAdding.java)

```
int sumListOfIntegers(List<Integer> data) {









}
```

Side note: Side Effects

- **Side Effect**: An observable change to state after code executes
  - Mutators, by definition, have side effects: they change data on their object.
  - However, some side effects are unexpected

- Expectation
  - Don't change the parameters you are passed unless purpose of a method.

```
void setDate(Date d) {
  d.setTime(0);
  this.date = d;
}
```

Iterable

Iterable

- The java Iterable interface (in java.lang package) is the root interface for the for-each loop
- The Iterable interface has only one method called iterator()
    - It returns must return an Iterator object which can be used to iterate the elements of the object implementing the Iterable interface

```
public interface Iterable<T> {
    public Iterator<T> iterator();
}
```

Iterable

- Ex: In a University's system, a `Degree` class stores a set of required `Courses`, and a set `Students` currently in the major. (Course.java) (Degree.java) (IterableDemo.java)
- Issues:

1. Semantically, it doesn't make sense that iterating over a major gives courses.
   - Why not iterate over:
     - Students?
     - Semesters?
2. Iterator has a remove() method!
   - What if I don't want allow others to remove objects?
3. What if I want to create my own iterator definitions?

Iterable Issues

Issue 1: Selecting the Iterator

- We can make a method in the `Degree` class that returns a `Iterable` object
- The client code can then request the `Iterable` object by name

Issue 2: Unmodifiable

- Prevent client code from modifying the list via the iterator's `remove()` method by using an unmodifiable view of your collection:

```
Collections.unmodifiableCollection(TYPE).iterator
```

Where `TYPE` is the Iterator type.

(Course.java) (Degree.java) (IterableDemo.java)

Iterable Issues

Issue 3: Custom Iterators

- Write your own iterators when needed.
- Implement `iterator()` function returning an iterator supporting `hasNext()` and `next()`.

- (Matrix.java) (MatrixTest.java)

Iterator Summary

- Use for-each loops when iterating over data.
- If your class has an obvious set of items to iterate over, implement `Iterable`
- If your class has non-obvious sets of items to iterate over, have methods that return `Iterable` objects
- Get iterators by just returning the iterator on your data structure:
  `return myArrayList.iterator();`
- Make unmodifiable views before returning an iterator:

```
return Collections.unmodifiableCollection(myArray).iterator();
```