Class Design Guidelines

Ch 3.1-3.4
1) Do we have choices for class design?
2) Why bother encapsulating data?
3) Can we combine an accessor and mutator?
Class Design Alternatives
Day Class

- **Task:** Design a Day class
  - Represent the year, month, and day of month.

- **Java provides the Date class**
  
  ```java
  Date now = new Date();
  System.out.println(now);  // calls..
  ```

- **Q:** What’s confusing about the Date class?
  -

- **How would we design our own class?**
Day Class

- **Class Responsibilities**
  - Able to work with a calendar day
  - Work in..
    (Not time, no time-zones...)

- **Public Interface**
  
  ```
  public class Day {
      public Day(int year, int month, int day);
      public int getYear();
      public int getMonth();
      public int getDate();
      public Day addDays(int n);
      public int daysFrom(Day other);
  }
  ```

  Calculate day in the future and “distance” between two days.
public class DayTester {
    public static void main(String[] args) {
        Day start = new Day(2050, 1, 31);
        System.out.println("Start: " + start);
        System.out.printf("Accessors: year %d, month %d, day %d.%n", 
                         start.getYear(), start.getMonth(), start.getDate());

        Day tomorrow = start.addDays(1);
        System.out.println("Tomorrow: " + tomorrow);

        Day future = start.addDays(1000);
        System.out.println("Future: " + future);

        int daysInFuture = future.daysFrom(start);
        System.out.println("Future is " + daysInFuture + " days away");
    }
}

Start: 2050-1-31
Accessors: year 2050, month 1, day 31.
Tomorrow: 2050-2-1
Future: 2052-10-28
Future is 1000 days away
• **Deprecated**
  - Parts of a public interface that are..
    - Usually means the deprecated part was not a good idea and has been redesigned.

• **Java's Date class similar to Day**
  - *Date* has many deprecated functions
    1. *getMonth()* should be avoided.
  - Use *Calendar* class instead.
  - Use built in Java classes when possible (here use *Calendar* instead of our *Day*).
Q: What's easy with this?

Q: What's hard?

- Days per month: 28, 30, 31
- Leap years; no year 0.

Efficiency
- Coded via `nextDay()`, `previousDay()`
- `myDay.addDays(10000)` runs 10,000 iterations!
Q: What's easy with this?

- public int daysFrom(DayTwo other) {
  return julian – other.julian;
}

Q: What's hard?

- (but not that complicated actually)

Efficiency:

System.out.printf("%d-%d-%d",
  d.getYear(), d.getMonth(), d.getDate());

- Have to do three conversions with fromJulian();
Day: Design 3

public class DayThree {
    private boolean ymdValid;
    private int year;
    private int month;
    private int date;
    private boolean julianValid;
    private int julian;

    // ... omitted
    public int getYear() {
        ensureYmd();
        return year;
    }
    public DayThree addDays(int n) {
        ensureJulian();
        // ... omitted
    }
}

- day number, and
- year/month/day.
- Lazy conversion: ..
  - If created via the day number, calculate year only when needed.
  - If created via year/month/day, calculate the day# when needed.
  - When a value is calculated..

- Functions check data validity:
  - If valid, then use it.
  - If invalid, calculate it & save answer.
Day: Design 3 (cont)

public class DayThree {
    private boolean ymdValid;
    private int year;
    private int month;
    private int date;
    private boolean julianValid;
    private int julian;
    // ... omitted
}

- **Q: What's easy?**
  - All code is...

- **Q: What's hard?**
  -

- **Q: What's the benefit of using lazy conversion and storing result?**
  - Only do the work when needed; only do the work once.

- **Q: What is the cost?**
  - Slightly more..
Day Design Summary

- **Implementations:**
  - **DayOne:** Work on year, month, day.
  - **DayTwo:** Work on a day's number (Julian day).
  - **DayThree:** Lazy conversion between both.

- **Which is best?**
  - Working with:
    - Year/Month/Day: **DayOne**
    - Julian days (addDays(),...): **DayTwo**
    - Efficiency: **DayThree**
      - Simplest code: not **DayThree**
Encapsulation
Ch 3.4
Encapsulation

- What's wrong with Day (on right)
  -

- Q: Why is this bad?
  - If we switched to lazy calculations, must access data through public methods (**DayThree**):
    Must convert use of public variables to methods:

```java
public class Day {
    public int year;
    public int month;
    public int day;
    // ... omitted.
}
```

```java
int year = myDay.year;
becomes
int year = myDay.year++;
```

```java
myDay.year++;
becomes
myDay = new Day(
    myDay.getYear() + 1,
    myDay.getMonth(),
    myDay.getDay());
```
Day Interface Design

- **Day Class's Interface**
  - The “helper” functions are private
  - **Ex:** `ensureJulian()`, `toJulian()`

- **Why keep helper methods private?**
  - able to change private details without having to re-write clients.
  - Expose only enough functionality to do the job!
Breaking Encapsulation

• Breaking encapsulation bad because..
  – What's hidden can change easily:
  – Seems overkill for small projects, but pays off on large projects.

• Benefits of Encapsulation
  – Reduces the amount a developer has to keep in mind at once:

Always code like your code matters.
Immutable

- **Immutable**: an object with..
  - Once created, you cannot change its (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..
  
  - Similar to `String.toLower()`:
    `String msg = "Hello World".toLowerCase();`
Why go Immutable?

- Avoids setter problems
  What day should this create?
  ```java
  Day start = new Day(2000, 1, 31);
  start.setMonth(2);
  ```
  - Feb 28?
  - Mar 3?
    - `setMonth()` would have to make an arbitrary choice on how to adjust the day to become valid.

- Shared reference
  - Cannot change behind your back.

- Thread-safe (later)
Shared Reference Problem

- **Client w/ Mutable Date:**
  - **Date** is *mutable* (supporting `setTime()`).
  - What's the problem with the following?

```java
public class Person {
    private Date birthDay;
    public Person(Date bDay) {
        birthDay = bDay;
    }
    public Date getBirthDay() {
        return birthDay;
    }
}

private static void exploitGetBirthDay() {
    Person george = new Person(new Date());
    System.out.println("Before: " + george.getBirthDay());
    Date date = george.getBirthDay();
    date.setTime(0);
    System.out.println("After: " + george.getBirthDay());
}
```
Clone() solution

- Protect Person from unexpected change:
  - Use an \textit{date} object; or
  - Use \texttt{clone()} to return a.. vs a reference to the original object.

```java
public class PersonWithClone {
    private Date birthDay;
    public PersonWithClone(Date birthDay) {
        this.birthDay = (Date) birthDay.clone();
    }

    public Date getBirthDay() {
        return (Date) birthDay.clone();
    }
}
```
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..

• Can be assigned a value either:
  a)..
  ```java
  private class Car {
      final private String MAKE = "PORCHE";
  }
  ```
  b)..
  ```java
  private class Truck {
      final private String MAKE;
      public Truck() {
          MAKE = "Ford";
      }
  }
  ```
public class Grade {
    public final int MAX_PERCENT = 100;
    private final ArrayList<Person> list;
    public Grade() {
        list = new ArrayList<Person>();
    }
}

Which generate compiler errors?

a)  

b)  

c)  

d)  

e)  

// Which of the following lines fail?
// 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(new Date()));
Command/Query Separation (Guideline)

A good idea; not a rule.
Command-Query Separation

- **Command**: A method which..
  (sometimes called a mutator)

- **Query**: A method which..
  (sometimes called an accessor)

**Command-Query Separation Guideline**:
Each method should do at most one of:
- Change state of an object.
- Return a value/part of the state.

**Q**: What is an object with no command methods?
Violation

• Example violation of Command-Query Separation
  public class BankAccount {
    private int balance = 0;
    public int getBalance(int value) {
      return balance -= value;
    }
  }

• Two required changes to fix:
  1.
  2. Don't write an actual getBalance().
Iterators

Iterators:

```java
public class IteratorExample {
    public static void main(String[] arg) {
        // Create the list
        List<String> data = new LinkedList<>();
        for (int i=0; i < 5; i++) {
            data.add("Value " + i);
        }

        // Standard for loop
        for (int i = 0; i < data.size(); i++) {
            System.out.printf("%d = %s%n", i, data.get(i));
        }

        // Iterator
        Iterator<String> itr = data.iterator();
        while (itr.hasNext()) {
            System.out.printf("%s%n", itr.next());
        }
    }
}
```

```java
interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

.interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```
Exercise

• Complete this function, using an iterator, to add up all numbers in the following collection:

```java
int sumListOfIntegers(List<Integer> data) {
    // Your code here
}
```
Iterators

- What violates command-query separation?
  - Individual methods for access (query/accessor) and change (command/mutator) often better.
    - Try to make commands (mutators) return `void`.

```java
public class IteratorExample {
    public static void main(String[] arg) {
        List<String> data = new LinkedList<>();

        // ... adding items omitted.
        Iterator<String> itr = data.iterator();
        while (itr.hasNext()) {
            System.out.printf("%s%n", itr.next());
        }
    }
}
```
Side Effects

• **Side Effect:**
  
  - Ex: \( x = 10; y++; \text{myDate}.\text{setTime}(0); \)
  
  - Mutators have side effects: they change data on their object.

• **Other possible side effects**
  
  -

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

```java
void setDate(Date date) {
    date.setTime(0);
    this.date = date;
}
```
What's wrong with this code trying to add up all positive numbers in the list?

```java
public class BadIteratorExample {
    public static void main(String[] arg) {
        List<Integer> data = new LinkedList<Integer>();

        // ... adding items omitted.

        int sum = 0;
        Iterator<Integer> itr = data.iterator();
        while (itr.hasNext()) {
            if (itr.next() >= 0) {
                ..
                sum += itr.next();
            }
        }
    }
}
```
How can custom classes support the for-each loop?

- **Ex:** In a recording Artist class stores a set of Song objects (among other things):

  ```java
  public boolean hasPlatinumSong(Artist artist) {
    for (Song song : artist) {
      if (song.isPlatinum()) {
        return true;
      }
    }
    return false;
  }
  ```
Iterable<T>

- for-each loop..
  (those that implement Iterable)
  ```java
  interface Iterable<T> {
    Iterator<T> iterator();
  }
  ```
- Make your collection classes implement Iterable!

```java
public class Artist implements Iterable<Song> {
  private List<Song> songs = new ArrayList<>();

  // Other functions omitted

  @Override
  public Iterator<Song> iterator() {
    return songs.iterator();
  }
}
```
Two Problems

- Does it make sense that iterating over an Artist gives Songs?
  - Why not iterate over an Artist for:
    - Albums?
    - Concerts?

- Iterator has a `remove()` method!
  - What if I don't want allow others to remove objects?
Selecting the Iterator

- Make a function that...

Client code can request the correct set of objects to iterate over by name.

```java
public class Artist {
    // Return Iterable objects:
    public Iterable<Song> songs() {
        return new Iterable<Song>() {
            @Override
            public Iterator<Song> iterator() {
                return songs.iterator();
            }
        };
    }

    public Iterable<Album> albums() {...}
    public Iterable<Concert> concerts() {...}
}

Usage in client code:
Artist bach = new Artist();
for (Album album : bach.albums()) {
    // use album here...
}
```
Unmodifiable

- Prevent client code from modifying the list via the iterator's `remove()` method by...

```java
public class Artist implements Iterable<Song>{
    private List<Song> songs = new ArrayList<>();

    @Override
    public Iterator<Song> iterator() {
        return Collections.unmodifiableCollection(songs).iterator();
    }
}
```

It actually creates a wrapper object that hides the underlying collection.
Write your own iterators when needed.

Implement `iterator()` function returning an iterator supporting `hasNext()` and `next()`.

```java
public class Matrix implements Iterable<Integer>{
    public static int NUM_ROWS;
    public static int NUM_COLS;
    private int[][] values;

    @Override
    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {
            int row = 0, col = 0;
            @Override
            public boolean hasNext() {
                return (row < NUM_ROWS) && (col < NUM_COLS);
            }
            @Override
            public Integer next() {
                Integer item = values[row][col];
                // ... code to advance col and row...
                return item;
            }
            @Override
            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
}
```
Iterator Advice

- Use **for-each** loops when iterating over data.
- If your class has an **obvious set** of items to iterate over.
- If your class has **non-obvious sets** of items to iterate over, have..
- Get most iterators by just returning the iterator on your data structure:
  ```java
  return myArrayList.iterator();
  ```
- Almost always make **unmodifiable views** before returning an iterator:
  ```java
  return Collections.unmodifiableCollection(myArray).iterator();
  ```
Summary

- **Three Day class design options**
  - **DayOne**: Work on year, month, day.
  - **DayTwo**: Work on a day's number (Julian day).
  - **DayThree**: Lazy conversion between both.

- **Encapsulation**: Limit scope of changes.

- **Immutable**: Visible state unchangeable
  - No shared reference problems.

- **Final fields**: Variable cannot be changed.

- **Command Query Separation**

- **Iterators and Iterable**