

Class Design Guidelines

Ch 3.1-3.4

Topics

- 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

```
Date now = new Date();  
System.out.println(now);           // calls date.toString()
```

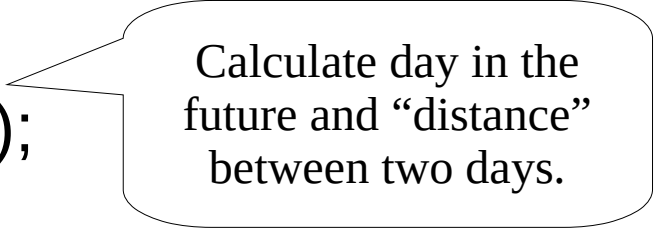
print out: Sun Feb 03 18:55:11 PST 2050
- Q: Whats confusing about the Date class?
 - named Date, but also represents Time
- How would we design our own class?

Day Class

- Class Responsibilities
 - Able to work with a calendar day
 - Work in days, months, years, or day numbers
(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.

Example Client Code

```
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

- **Deprecated**
 - Parts of a public interface that are..
no longer supported or recommended
 - 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
Ex: getMonth() should be avoided.
 - Use LocalDate or LocalDateTime class instead.
 - Use built in Java classes when possible
(here use LocalDate instead of our Day).

Day: Design 1

```
public class DayOne {
    private int year;
    private int month;
    private int date;

    public DayOne(int year,
                  int month, int date) {
        this.year = year;
        this.month = month;
        this.date = date;
    }

    public int getYear() {
        return year;
    }

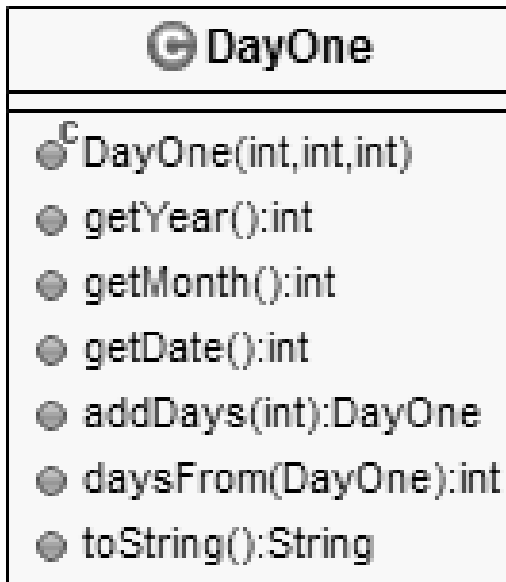
    private DayOne nextDay() {
        // .. omitted.
    }
    // ... omitted
}
```

- store year, month, day as fields
- Q: What's easy with this?
constructors, accessors
- Q: What's hard?
addDays(), daysFrom()
 - Days per month: 28, 30, 31
 - Leap years; no year 0.
- Efficiency
 - Coded via nextDay(), previousDay()
 - myDay.addDays(10000)
runs 10,000 iterations!

Day: Design 2

Store day as a **day number since a fixed start day**

```
public class DayTwo {  
    // Store the "Julian" day number.  
    private int julian;  
  
    //... omitted.  
}
```



- Q: What's easy with this?
 - **addDays(), daysFrom()**
public int daysFrom(DayTwo other) {
 return julian – other.julian;
}
- Q: What's hard?
 - **constructor, accessors: getYear()**
(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()!

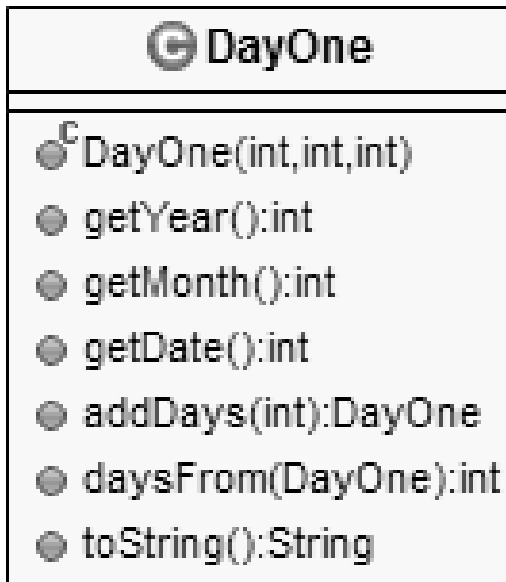
Day: Design 3

```
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  
    }  
}
```

- **store both** day number, and year/month/day.
- **Lazy conversion**: **calculate when needed**
 - 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..
cache it for future use
- 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..
reasonably straight forward
- Q: What's hard?
 - extra work maintaining the valid-flags
- Q: What's the benefit of using lazy conversion and storing result?
 - efficiency: aka cache
Only do the work when needed;
only do the work once.
- Q: What is the cost?
 - Slightly more.complicated code
More space

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?
depends on the application
 - 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)

- directly exposes data

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

- 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:

`int year = myDay.year;`

`myDay.year++;`

becomes

becomes














`int year = myDay.getYear();`

`myDay = new Day(
 myDay.getYear() + 1,
 myDay.getMonth(),
 myDay.getDay());`

Day Interface Design

note: we want as much as private as possible

- Day Class's Interface
 - The “helper” functions are private
 - Ex: ensureJulian(), toJulian()
- Why keep helper methods private?
 - **Encapsulation:**
able to change private details without having to re-write clients.
 - Expose only enough functionality to do the job!

<<Java Class>>	
 DayThree (default package)	
<ul style="list-style-type: none">▣ year: int▣ month: int▣ date: int▣ ymdValid: boolean▣ julianValid: boolean▣ julian: int	
 ^C DayThree(int,int,int)	 ^C DayThree(int)
 getYear():int	
 getMonth():int	
 getDate():int	
 addDays(int):DayThree	
 daysFrom(DayThree):int	
 toString():String	
 ensureJulian():void	
 ensureYmd():void	
 ^S toJulian(int,int,int):int	
 ^S fromJulian(int):int[]	

Breaking Encapsulation

- Breaking encapsulation bad because..
extensive changes inhibit making updates
 - What's hidden can change easily..
promotes refactoring
 - Seems overkill for small projects, but pays off on large projects.

Always code like your code matters.

- Benefits of Encapsulation
 - reduces the scope of a change
 - Reduces the amount a developer has to keep in mind at once: reduced cognitive load

Immutable

- Immutable: an object with *no methods that change its visible state*
 - Once created, you cannot change its (visible) state.
- Q: Is DayThree immutable?
 - Lazy conversion changes its private fields.
 - *it's immutable:*
externally it has the same state.
- Immutability implications for Day
 - addDays() must return *a new Day object*
 - Similar to String.toLowerCase():
String msg = "Hello World".toLowerCase();

Why go Immutable?

- Avoids setter problems

What day should this create?

```
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?

```
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);  
    client changes george's birthday  
    System.out.println(  
        "After: " + george.getBirthDay());  
}
```

problem here: shared reference (birthday):

- getBirthDay return a reference to birthday field
- later on the client calls setTime on that reference

Clone() solution

java copy constructor

- Protect Person from unexpected change:
 - Use an `immutable` date object; or
 - Use `clone()` to return a `duplicate object` vs a reference to the original object.

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

`clone()` has return type of Object
--> need to cast to Date

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)
No: shared reference
 - a reference to a field of a immutable type? (Ex: String)
Yes: cannot change the object (String is an immutable — can not be changed)
 - a primitive typed field? (Ex: int)
Yes: pass by value
- 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 *final and immutable are not the same, but similar

- A field can be marked final meaning..

variable cannot be made to reference another object
(or change its value if a primitive)

- Can be assigned a value either:

a).when declared

```
private class Car {  
    final private String MAKE = "PORCHE";  
}
```

b).once during the constructor

```
private class Truck {  
    final private String MAKE;  
    public Truck() {  
        MAKE = "Ford";  
    }  
}
```

final Example

```
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) No bcuz int will pass by value
- b) Yes
- c) Yes, bcuz we're trying to change reference to new object
- d) No
- e) No bcuz object is still mutable

// ... cont...

```
public void doSomething() {  
    // 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()));  
}
```

}

Note: mutability is not something deal with keyword,
it's about how u design
bcuz only found mutable or not at compile time

Command/Query Separation (Guideline)



A good idea;
not a rule.

Command-Query Separation

- Command: A method which.. *changes an object*
(sometimes called a mutator)
- Query: A method which..
returns the state of an object without changing it
(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?
 - *immutable*

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. rename to `withdraw()`

when doing code review, think about:
shared reference

2. Don't `return the value`

—> should follow principle of least surprise

write an actual `getBalance()`.

Iterators

- Iterators: [abstract iteration over a data set](#)

```
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());  
        }  
    }  
}
```

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

.iterator() returns an..
[Iterator object](#)

Iterator is a generic.

[next\(\)](#) returns next
element and advances

-> [violate command query](#)

Exercise

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

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

```
    Iterator<Integer> itr = data.iterator();
```

```
    int sum = 0;
```

```
    while (itr.hasNext()) {
```

```
        sum += itr.next();
```

```
    }
```

```
    return sum;
```

```
}
```

Iterators

- What violates command-query separation?
 - `itr.next()`: moves to next, AND reads state

```
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());  
        }  
    }  
}
```

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

Side Effects

- Side Effect: *an observable change to state after code executes*
 - Ex: `x = 10; y++; myDate.setTime(0);`
 - Mutators have side effects: they change data on their object.
- Other possible side effects
 - *change parameter unexpectedly*
- Expectation
 - Don't change the parameters you are passed unless purpose of a method.

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

Bad Code Example

- What's wrong with this code trying to add up all positive numbers in the list?

```
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();  
            }  
        }  
    }  
}
```

common bug:
calling next() more than once

Iterable

Adding for-each support

- How can custom classes support the for-each loop?
 - Ex: In a recording Artist class stores a set of Song objects (among other things):

Inside Main class:

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

Iterable<T>

- for-each loop.[works on Iterable objects](#)
(those that implement Iterable)

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

- Make your collection classes implement Iterable!

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

```
// Other functions omitted
```

```
@Override
```

```
public Iterator<Song> iterator() {  
    return songs.iterator();  
}
```

```
}
```

[reason why we have an iterator here:](#)

[-external code needs to access list of songs](#)

[+if we make the list to be public -> violate encapsulation](#)

[+"make the class iterable" \(overrides iterator method\) could return the list](#)

[-> allow external code to interact with the song list](#)

[-> not the best option but still a good start](#)

Two Problems with Iterator

- 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

solution:

can make these shorter by using lambda expressions

- Make a function that..
return an anonymous Iterable object

```
public class Artist {  
    // Return Iterable objects:  
    public Iterable<Song> songs() {  
        return new Iterable<Song>() {  
            @Override return an anno class and anno object  
            public Iterator<Song> iterator() {  
                return songs.iterator();  
            }  
        };  
    }  
  
    public Iterable<Album> albums() {...}  
    public Iterable<Concert> concerts() {...}  
}
```

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

```
Usage in client code:  
Artist bach = new Artist();  
for (Album album : bach.albums()) {  
    // use album here...  
}  
albums() could be thought of static factor method
```

Unmodifiable

solution for problem 2:

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

```
public class Artist implements Iterable<Song>{  
    private List<Song> songs = new ArrayList<>();  
  
    @Override  
    public Iterator<Song> iterator() {  
        return Collections.unmodifiableCollection(songs).iterator();  
    }  
}
```

this is a static method that:

- creates a wrapper object sit on top of the real song list
- it will override the remove method in iterator (just throwing exception)

It actually creates a wrapper object that hides the underlying collection.

Custom Iterator

Write your own
iterators when
needed.

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

```
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
`implement Iterable`
- If your class has non-obvious sets of items to iterate over, have `methods that return Iterable objects`
- Get most iterators by just returning the iterator on your data structure:
`return myArrayList.iterator();`
- Almost always make unmodifiable views before returning an iterator:
`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