Class design guidelines
Encapsulation

• Classes can be implemented many different ways
  – each has advantages & disadvantages
  – improvement/revision always a possibility
• Encapsulation facilitates change in implementation without affecting client code
Information Hiding

• Public interface of object & private representation of object usually kept distinct
  – allows certain aspects of object to be hidden from view
  – distinguishes ability to perform a task from specific steps needed to do so
  – public interface of object consists of operations other objects can request the object to perform
Why private data fields?

• Public instance fields box us in
  – anything public can be, and probably will be, used by client programmers
  – any changes made to class must accommodate this dependency

• For example, if the Day class had public members representing month, day and year, a switch to julian representation would involve either invalidating this code or significant (and cumbersome) redundancy
The Importance of Encapsulation

- Even a simple class can benefit from different implementations
- Users are unaware of implementation
- Public instance variables would have blocked improvement in Day class
Day class example - public data fields year, day, month

- Suppose we switch to julian representation
- Can't just wave magic wand to replace all `d.year` with `d.getYear()` in client code
- How about `d.year++`?
  
  ```java
d = new Day(d.getDay(), d.getMonth(), d.getYear() + 1)
```

- Gets really inefficient in Julian representation
- Don't use public fields, even for "simple" classes
Advantages of encapsulation

• Restricts range of program affected by any change to the methods of the class
• Allows for software evolution over time without useless baggage carried into next program version
Accessors & mutators

• Accessor methods report but do not change object state
• Mutator methods modify object state
• Classes without mutators are immutable
  – String class is an example; can assign a new object reference to a String variable, but can’t make changes to the object
Don't Supply a Mutator for every Accessor

- Day has `getYear`, `getDate` and `getMonth` accessors, but does *not* have `setYear`, `setMonth`, `setDate` mutators

- These mutators would not work well; for example:

  ```java
  Day deadline = new Day(2001, 1, 31);
  deadline.setMonth(2); // ERROR
  deadline.setDate(28);
  ```
Example continues

- Maybe we should call `setDate` first?

  Day d = new Day(2001, 2, 28);
  d.setDate(31); // ERROR
  d.setMonth(3);

- `GregorianCalendar` implements confusing `rollover` - silently gets the wrong result instead of error:
  - suppose date initially set to January 31, then month set to February
  - instead of error message, get date set to March 3
Sharing Mutable References

- References to immutable objects can be freely shared
- Don't share mutable references: example

```java
class Employee {
    . . .
    public String getName() { return name; }
    public double getSalary() { return salary; }
    public Date getHireDate() { return hireDate; }
    // note: Date objects are mutable
    . . .
}
```
Example continued

• Pitfall:

```java
Employee harry = . . .;
Date d = harry.getHireDate();
d.setTime(t); // changes Harry's state
```

• Remedy: Use clone()

```java
public Date getHireDate()
{
    return (Date)hireDate.clone();
}
```
Final Instance Fields

- Good idea to mark immutable instance fields as final:
  
  ```java
  private final int day;
  ```

- final object reference can still refer to mutating object:
  
  ```java
  private final ArrayList elements;
  ```

  - elements can't refer to another array list
  - The contents of the array list can change
Separating Accessors and Mutators

• If we call a method to access an object, we don't expect the object to mutate
• Rule of thumb: Mutators should return `void`
• Example of violation (from Java API):
  ```java
  StringTokenizer t = . . .;
  String s = t.nextToken();
  ```
  Yields current token *and* advances iteration - what if I want to read the current token again?
Separating Accessors and Mutators

• Better interface:
  ```java
  String getToken();
  void nextToken();
  ```

• Even more convenient:
  ```java
  String getToken();
  String nextToken(); // returns current
  ```

• Refine rule of thumb: Mutators can return a convenience value, provided there is also an accessor to get the same value
Side Effects

- Accessor: no change to object
- Mutator: changes object state
- Side effect: change to another object
  - Parameter variable
  - static object
- Avoid side effects--they confuse users
Side Effects

• Avoid modifying static objects
• Example: don't print error messages to System.out:
  
  ```java
  if (newMessages.isFull())
    System.out.println("Sorry--no space");
  ```

• Your classes may need to run in an environment without System.out; throw exception instead
Law of Demeter

• A method should only use objects that are
  – instance fields of its class
  – parameters
  – objects that it constructs with new

• Shouldn't use an object that is returned from a method call

• A method should not ask another object to give it part of its internal state to work on
Analyzing Quality of Class Interface

• Customers: Programmers using the class
• Criteria:
  – Cohesion
  – Completeness
  – Convenience
  – Clarity
  – Consistency
• Engineering activity: make tradeoffs
Cohesion

- Class describes a *single* abstraction
- Methods should be related to the single abstraction
- If a class has unrelated responsibilities, split it up
Completeness

• Support all operations that are well-defined on abstraction

• Potentially bad example: `Date`

  ```java
  Date start = new Date();
  // do some work
  Date end = new Date();
  ```

• How many milliseconds have elapsed? - No such operation in `Date` class

• Does it fall outside the responsibility?
Convenience

- A good interface makes all tasks possible, and common tasks simple

- Bad example: Reading from `System.in`
  - Why doesn't `System.in` have a `readLine` method analogous to `System.out.println`?
  - Why can't I make a `BufferedReader` from an input stream when I can make a `PrintWriter` from an output stream?
Clarity

- Confused programmers write buggy code
- Bad example: Removing elements from LinkedList:
  ```java
  LinkedList countries = new LinkedList();
countries.add("A");
countries.add("B");
countries.add("C");
  ```
- Iterate through list:
  ```java
  ListIterator iterator =
countries.listIterator();
  while (iterator.hasNext())
    System.out.println(iterator.next());
  ```
Bad example continued

- Iterator *between* elements, like blinking caret in word processor; `add()` adds to the left of iterator (like word processor):

  ```java
  ListIterator iterator = countries.listIterator(); // |ABC
  iterator.next(); // A|BC
  iterator.add("X"); // AX|BC
  ```

- To remove first two elements, you can't just "backspace"
Bad example continued

- `remove()` does *not* remove element to the left of iterator

- From API documentation:
  
  Removes from the list the last element that was returned by next or previous. This call can only be made once per call to next or previous. It can be made only if add has not been called after the last call to next or previous.

- Huh?
Consistency

• Related features of a class should have matching:
  • names
  • parameters
  • return values
  • behavior

• Bad example:
  new GregorianCalendar(year, month - 1, day)

• Why is month 0-based?
Programming by Contract

• Spell out responsibilities:
  • of caller
  • of implementor
• Increase reliability
• Increase efficiency
Preconditions

• Excessive error checking is costly
• Returning dummy values can complicate testing
• Contract metaphor
  – Service provider must *specify* preconditions
  – If precondition is fulfilled, service provider must work correctly
  – Otherwise, service provider can do *anything*
Preconditions

- When precondition fails, service provider may
  - throw exception
  - return false answer
  - corrupt data
Assertions

• Mechanism for warning programmers; can be turned off after testing

• Syntax:
  
  assert condition;  //note: no parentheses
  assert condition : explanation;

• Throws AssertionError if condition false and checking enabled

• During testing, run with
  
  java -enableassertions MyProg
Exceptions in the contract

- Exception throw part of the contract
- Caller can *rely* on behavior
- Exception throw *not result of precondition violation*
- This method has *no* precondition
Example

/**
 * @throws IllegalArgumentException
 * if queue is empty
 */

public Message removeFirst()
{
    if (count == 0)
        throw new IllegalArgumentException();
    Message r = elements[head];
    . . .
Postconditions

• Conditions that the service provider guarantees
• Every method promises description, @return
• Sometimes, can assert additional useful condition
Class Invariants

• Condition that is
  – true after every constructor (no invalid objects can be created)
  – preserved by every method (if it's true before the call, it's again true afterwards)
  – Useful for checking validity of operations
Class invariants

• As long as instance fields are private, you have complete control over object modification
• Can guarantee certain values are within legal range, certain references are not null
• Invariant is appropriate tool for documenting such a guarantee
Interface vs. Implementation

Invariants

- Interface invariants are conditions that involve only public interface of class
  - of interest to class user
  - give behavior guarantee for class objects

- Implementation invariants involve details of particular implementation
  - of interest to class implementor
  - can be used to ensure implementation correctness