Object-Oriented Design & Patterns
Cay S. Horstmann

Chapter 3
Guidelines for Class Design
Chapter Topics

• An overview of the Date classes in the Java library
• Designing a Day class
• Three implementations of the Day class
• The importance of encapsulation
• Analyzing the quality of an interface
• Programming by contract
• Unit testing
• Many programs manipulate dates such as "Saturday, February 3, 2001"

• Date class:
  Date now = new Date();
  // constructs current date/time
  System.out.println(now.toString());
  // prints date such as
  // Sat Feb 03 16:34:10 PST 2001

• Date class encapsulates point in time
Methods of the Date class

• boolean after(Date other)
  Tests if this date is after the specified date

• boolean before(Date other)
  Tests if this date is before the specified date

• int compareTo(Date other)
  Tells which date came before the other

• long getTime()
  Returns milliseconds since the epoch
  (1970-01-01 00:00:00 GMT)

• void setTime(long n)
  Sets the date to the given number of milliseconds since the epoch
Methods of the Date class

- Deprecated methods omitted
- Date class methods supply total ordering on Date objects
- Convert to scalar time measure
- Note that before/after not strictly necessary
- (Presumably introduced for convenience)
Points in Time

Big Bang

Epoch = 1970-01-01 00:00:00 GMT

Points in time represented by two Date objects
The GregorianCalendar Class

- The Date class doesn't measure months, weekdays, etc.
- That's the job of a calendar
- A calendar assigns a name to a point in time
- Many calendars in use:
  - Gregorian
  - Contemporary: Hebrew, Arabic, Chinese
  - Historical: French Revolutionary, Mayan
Date Handling in the Java Library

Date

Gregorian Calendar

Calendar
Designing a Day Class

• Custom class, for teaching/learning purpose

• Use the standard library classes, not this class, in your own programs

• Day encapsulates a day in a fixed location

• No time, no time zone

• Use Gregorian calendar
Designing a Day Class

• Answer questions such as:

How many days are there between now and the end of the year?

What day is 100 days from now?
Designing a Day Class

<table>
<thead>
<tr>
<th>Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>relate calendar days to day counts</td>
</tr>
</tbody>
</table>
Designing a Day Class

• daysFrom computes number of days between two days:

```java
int n = today.daysFrom(birthday);
```

• addDays computes a day that is some days away from a given day:

```java
Day later = today.addDays(999);
```

• Mathematical relationship: addDays() and daysFrom() are inverses

```java
d.addDays(n).daysFrom(d) == n
d1.addDays(d2.daysFrom(d1)) == d2
```

• Clearer when written with "overloaded operators": (Not available in Java)

```java
(d + n) - d == n
d1 + (d2 - d1) == d2
```

• Constructor Date(int year, int month, int date)

• getYear, getMonth, getDate acccessors
Implementing a Day Class

- Straightforward implementation:
  ```java
  private int year
  private int month
  private int date
  ```
- `addDays/daysBetween` tedious to implement
  - April, June, September, November have 30 days
  - February has 28 days, except in leap years it has 29 days
  - All other months have 31 days
- Leap years are divisible by 4, except after 1582, years divisible by 100 but not 400 are not leap years
- There is no year 0; year 1 is preceded by year -1
- In the switchover to the Gregorian calendar, ten days were dropped:
  October 15, 1582 is preceded by October 4
Implementing a Day Class

001: public class Day
002: {
003:     /**
004:      Constructs a day with a given year, month, and day
005:      of the Julian/Gregorian calendar. The Julian calendar
006:      is used for all days before October 15, 1582
007:      @param aYear a year != 0
008:      @param aMonth a month between 1 and 12
009:      @param aDate a date between 1 and 31
010:      */
011:     public Day(int aYear, int aMonth, int aDate)
012:     {
013:         year = aYear;
014:         month = aMonth;
015:         date = aDate;
016:     }
Implementing a Day Class

- 018: /**
- 019: Returns the year of this day
- 020: @return the year
- 021: */
- 022: public int getYear()
  
  023: {
  024:     return year;
  025: }
  
  026:
  027: /**
  028: Returns the month of this day
  029: @return the month
  030: */
  031: public int getMonth()
  
  032: {
  033:     return month;
  034: }
  
  035:
Implementing a Day Class

• 036: /**
• 037: Returns the day of the month of this day
• 038: @return the day of the month
• 039: */
• 040: public int getDate()
• 041: {
• 042:     return date;
• 043: }
• 044:
Implementing a Day Class

045: /**
046: Returns a day that is a certain number of days away from
047: this day
048: @param n the number of days, can be negative
049: @return a day that is n days away from this one
050: */
051: public Day addDays(int n)
052: {
053:     Day result = this;
054:     while (n > 0)
055:     {
056:         result = result.nextDay();
057:         n--;
058:     }
059:     while (n < 0)
060:     {
061:         result = result.previousDay();
062:         n++;
063:     }
064:     return result;
065: }
066: }
Implementing a Day Class

- 067: /**
- 068: Returns the number of days between this day and another
- 069: day
- 070: @param other the other day
- 071: @return the number of days that this day is away from
- 072: the other (>0 if this day comes later)
- 073: */
- 074: public int daysFrom(Day other)
- 075: {
- 076:     int n = 0;
- 077:     Day d = this;
- 078:     while (d.compareTo(other) > 0)
- 079:     {
- 080:         d = d.previousDay();
- 081:         n++;
- 082:     }
- 083:     while (d.compareTo(other) < 0)
- 084:     {
- 085:         d = d.nextDay();
- 086:         n--;
- 087:     }
- 088:     return n;
- 089: }
Implementing a Day Class

- 091: /**
- 092:  Compares this day with another day.
- 093:  @param other the other day
- 094:  @return a positive number if this day comes after the
- 095:  other day, a negative number if this day comes before
- 096:  the other day, and zero if the days are the same
- 097:  */
- 098:  private int compareTo(Day other)
- 099:  {
- 100:      if (year > other.year) return 1;
- 101:      if (year < other.year) return -1;
- 102:      if (month > other.month) return 1;
- 103:      if (month < other.month) return -1;
- 104:      return date - other.date;
- 105:  }
- 106:
/**
 * Computes the next day.
 * @return the day following this day
 */

private Day nextDay()
{
    int y = year;
    int m = month;
    int d = date;

    if (y == GREGORIAN_START_YEAR
        && m == GREGORIAN_START_MONTH
        && d == JULIAN_END_DAY)
        d = GREGORIAN_START_DAY;
    else if (d < daysPerMonth(y, m))
        d++;
    else
    {
        d = 1;
        m++;
        if (m > DECEMBER)
        {
            m = JANUARY;
            y++;
            if (y == 0) y++;
        }
    }

    return new Day(y, m, d);
}
Implementing a Day Class

• 137: /**
• 138:  Computes the previous day.
• 139:  @return the day preceding this day
• 140: */
• 141: private Day previousDay()
• 142: { int y = year;
• 143:  int m = month;
• 144:  int d = date;
• 145:  
• 146:  
• 147:   if (y == GREGORIAN_START_YEAR
• 148:     && m == GREGORIAN_START_MONTH
• 149:     && d == GREGORIAN_START_DAY)
• 150:     d = JULIAN_END_DAY;
• 151:  else if (d > 1)
• 152:     d--;
• 153:  else
• 154:   { m--;
• 155:     if (m < JANUARY)
• 156:     { m = DECEMBER;
• 157:       y--;
• 158:       if (y == 0) y--;
• 159:     }
• 160:     d = daysPerMonth(y, m);
• 161:   }
• 162:   return new Day(y, m, d);
• 163: }
• 164: 
• 165: }
Implementing a Day Class

• 167: /**
• 168: Gets the days in a given month
• 169: @param y the year
• 170: @param m the month
• 171: @return the last day in the given month
• 172: */
• 173: private static int daysPerMonth(int y, int m)
• 174: {
• 175:     int days = DAYS_PER_MONTH[m - 1];
• 176:     if (m == FEBRUARY && isLeapYear(y))
• 177:         days++;
• 178:     return days;
• 179: }
• 180:
Implementing a Day Class

• 181: /**
• 182:   Tests if a year is a leap year
• 183:   @param y the year
• 184:   @return true if y is a leap year
• 185: */
• 186: private static boolean isLeapYear(int y)
• 187: { if (y % 4 != 0) return false;
• 189: if (y < GREGORIAN_START_YEAR) return true;
• 190: return (y % 100 != 0) || (y % 400 == 0);
• 191: }
• 193: private int year;
• 194: private int month;
• 195: private int date;
• 197: private static final int[] DAYS_PER_MONTH = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
• 200: private static final int GREGORIAN_START_YEAR = 1582;
• 201: private static final int GREGORIAN_START_MONTH = 10;
• 202: private static final int GREGORIAN_START_DAY = 15;
• 203: private static final int JULIAN_END_DAY = 4;
• 205: private static final int JANUARY = 1;
• 206: private static final int FEBRUARY = 2;
• 207: private static final int DECEMBER = 12;
• 208: }
public class DayTester {
    public static void main(String[] args) {
        Day today = new Day(2001, 2, 3); // February 3, 2001
        Day later = today.addDays(999);
        System.out.println(later.getYear() + "-" + later.getMonth() + "-" + later.getDate());
        System.out.println(later.daysFrom(today)); // Prints 999
    }
}

Note private helper methods: addDays() and daysFrom()

Helper methods should not be public

Computations are inefficient: a day at a time
Second Implementation: Julian Dates

- For greater efficiency, use Julian day number

- Used in astronomy

- Number of days since Jan. 1, 4713 BCE

- May 23, 1968 = Julian Day 2,440,000

- Greatly simplifies date arithmetic
Second Implementation: Julian Dates

001: public class Day
002: {
003:     /**
004:         Constructs a day with a given year, month, and day of the Julian/Gregorian calendar. The Julian calendar is used for all days before October 15, 1582
005:     */
006:     @param aYear a year != 0
007:     @param aMonth a month between 1 and 12
008:     @param aDate a date between 1 and 31
009:     */
010:     public Day(int aYear, int aMonth, int aDate)
011:     {
012:         julian = toJulian(aYear, aMonth, aDate);
013:     }
014: }
015: /**
016:     Returns the year of this day
017:     */
018:     public int getYear()
019:     {
020:         return fromJulian(julian)[0];
021:     }
Second Implementation: Julian Dates

025: /**
026:   Returns the month of this day
027:   @return the month
028: */
029: public int getMonth()
030: {
031:   return fromJulian(julian)[1];
032: }
033:
034: /**
035:   Returns the day of the month of this day
036:   @return the day of the month
037: */
038: public int getDate()
039: {
040:   return fromJulian(julian)[2];
041: }
042:
/**
   * Returns a day that is a certain number of days away from this day
   * @param n the number of days, can be negative
   * @return a day that is n days away from this one
   */
   public Day addDays(int n)
   {
      return new Day(julian + n);
   }

   /**
   * Returns the number of days between this day and another day.
   * @param other the other day
   * @return the number of days that this day is away from the other (>0 if this day comes later)
   */
   public int daysFrom(Day other)
   {
      return julian - other.julian;
   }

   private Day(int aJulian)
   {
      julian = aJulian;
   }
Second Implementation: Julian Dates

• 070: /**
• 071: Computes the Julian day number of the given day.
• 072: @param year a year
• 073: @param month a month
• 074: @param date a day of the month
• 075: @return The Julian day number that begins at noon of
• 076: the given day
• 077: Positive year signifies CE, negative year BCE.
• 078: Remember that the year after 1 BCE was 1 CE.
• 079: 
• 080: A convenient reference point is that May 23, 1968 noon
• 081: is Julian day number 2440000.
• 082: 
• 083: Julian day number 0 is a Monday.
• 084: 
• 085: This algorithm is from Press et al., Numerical Recipes
• 087: */
Second Implementation: Julian Dates

private static int toJulian(int year, int month, int date)
{
    int jy = year;
    if (year < 0) jy++;
    int jm = month;
    if (month > 2) jm++;
    else {
        jy--;
        jm += 13;
    }

    int jul = (int) Math.floor(365.25 * jy) + Math.floor(30.6001 * jm) + date + 1720995.0;
    int IGREG = 15 + 31 * (10 + 12 * 1582);
    // Gregorian Calendar adopted Oct. 15, 1582
    if (date + 31 * (month + 12 * year) >= IGREG)
        // Change over to Gregorian calendar
        {
            int ja = (int) (0.01 * jy);
            jul += 2 - ja + (int) (0.25 * ja);
        }

    return jul;
}
/**
 * Converts a Julian day number to a calendar date.
 * This algorithm is from Press et al., Numerical Recipes in C, 2nd ed., Cambridge University Press 1992
 * 
 * @param j the Julian day number
 * @return an array whose 0 entry is the year, 1 the month, and 2 the day of the month.
 */
private static int[] fromJulian(int j)
{
    int ja = j;
    int JGREG = 2299161;
    // The Julian day number of the adoption of the Gregorian calendar

if (j >= JGREG)
    // Cross-over to Gregorian Calendar produces this correction
    {
        int jalpha = (int) (((float) (j - 1867216) - 0.25) / 36524.25);
        ja += 1 + jalpha - (int) (0.25 * jalpha);
    }
int jb = ja + 1524;
int jc = (int) (6680.0 + ((float) (jb - 2439870) - 122.1) / 365.25);
int jd = (int) (365 * jc + (0.25 * jc));
int je = (int) ((jb - jd) / 30.6001);
int date = jb - jd - (int) (30.6001 * je);
int month = je - 1;
if (month > 12) month -= 12;
int year = jc - 4715;
if (month > 2) --year;
if (year <= 0) --year;
return new int[] { year, month, date };
}
Second Implementation: Julian Dates

```java
public class DayTester {
    public static void main(String[] args) {
        Day today = new Day(2001, 2, 3); // February 3, 2001
        Day later = today.addDays(999);
        System.out.println(later.getYear());
        System.out.println(later.getMonth());
        System.out.println(later.getDate());
        System.out.println(later.daysFrom(today)); // Prints 999
    }
}
```
Third Implementation

• Now constructor, accessors are inefficient
• Best of both worlds: Cache known Julian, y/m/d values

001: public class Day
002: {
003:  /**
004:   Constructs a day with a given year, month, and day
005:   of the Julian/Gregorian calendar. The Julian calendar
006:   is used for all days before October 15, 1582
007:   @param aYear a year != 0
008:   @param aMonth a month between 1 and 12
009:   @param aDate a date between 1 and 31
010:  */

public Day(int aYear, int aMonth, int aDate)
{
    year = aYear;
    month = aMonth;
    date = aDate;
    ymdValid = true;
    julianValid = false;
}

/**
 * Returns the year of this day
 * @return the year
 */
public int getYear()
{
    ensureYmd();
    return year;
}
Third Implementation

030: /**
031:  Returns the month of this day
032:  @return the month
033: */
034: public int getMonth()
035: {
036:   ensureYmd();
037:   return month;
038: }
039:
040: /**
041:  Returns the day of the month of this day
042:  @return the day of the month
043: */
044: public int getDate()
045: {
046:   ensureYmd();
047:   return date;
048: }
Third Implementation

• 050: /**
• 051:   Returns a day that is a certain number of days away from
• 052:   this day
• 053:   @param n the number of days, can be negative
• 054:   @return a day that is n days away from this one
• 055: */
• 056: public Day addDays(int n)
• 057: { ensureJulian();
• 059:     return new Day(julian + n);
• 060: }
• 062: /**
• 063:   Returns the number of days between this day and another
• 064:   day
• 065:   @param other the other day
• 066:   @return the number of days that this day is away from
• 067:   the other (>0 if this day comes later)
• 068: */
• 069: public int daysFrom(Day other)
• 070: {
• 071:    ensureJulian();
• 072:    other.ensureJulian();
• 073:    return julian - other.julian;
• 074: }
private Day(int aJulian) {
    julian = aJulian;
    ymdValid = false;
    julianValid = true;
}

/**
 * Computes the Julian day number of this day if necessary
 */
private void ensureJulian() {
    if (julianValid) return;
    julian = toJulian(year, month, date);
    julianValid = true;
}
Third Implementation

094:  /**
095:  Converts this Julian day number to a calendar date if necessary.
096:  */
097:  private void ensureYmd() {
099:     if (ymdValid) return;
100:     int[] ymd = fromJulian(julian);
101:     year = ymd[0];
102:     month = ymd[1];
103:     date = ymd[2];
104:     ymdValid = true;
105: }
/**
 * Computes the Julian day number of the given day.
 * 
 * @param year a year
 * @param month a month
 * @param date a day of the month
 * @return The Julian day number that begins at noon of the given day
 * Positive year signifies CE, negative year BCE.
 * Remember that the year after 1 BCE is 1 CE.
 * A convenient reference point is that May 23, 1968 noon is Julian day number 2440000.
 * Julian day number 0 is a Monday.
 * This algorithm is from Press et al., Numerical Recipes in C, 2nd ed., Cambridge University Press 1992
 */
private static int toJulian(int year, int month, int date)
{
    int jy = year;
    if (year < 0) jy++;
    int jm = month;
    if (month > 2) jm++;
    else
    {
        jy--;
        jm += 13;
    }
    int jul = (int) (java.lang.Math.floor(365.25 * jy) + java.lang.Math.floor(30.6001 * jm) + date + 1720995.0);
    int IGREG = 15 + 31 * (10 + 12 * 1582);
    // Gregorian Calendar adopted Oct. 15, 1582
    if (date + 31 * (month + 12 * year) >= IGREG)
    // Change over to Gregorian calendar
    {
        int ja = (int) (0.01 * jy);
        jul += 2 - ja + (int) (0.25 * ja);
    }
    return jul;
}
Third Implementation

152:    /**
153:       Converts a Julian day number to a calendar date.
154:
155:       This algorithm is from Press et al., Numerical Recipes in C, 2nd ed., Cambridge University Press 1992
157:
158:       @param j  the Julian day number
159:       @return an array whose 0 entry is the year, 1 the month,
160:          and 2 the day of the month.
161:    */
private static int[] fromJulian(int j) {
    int ja = j;
    int JGREG = 2299161;
    // The Julian day number of the adoption of the Gregorian calendar
    if (j >= JGREG) {
        // Cross-over to Gregorian Calendar produces this correction
        int jalpha = (int) (((float) (j - 1867216) - 0.25) / 36524.25);
        ja += 1 + jalpha - (int) (0.25 * jalpha);
    }
    int jb = ja + 1524;
    int jc = (int) (6680.0 + ((float) (jb - 2439870) - 122.1) / 365.25);
    int jd = (int) (365 * jc + (0.25 * jc));
    int je = (int) ((jb - jd) / 30.6001);
    int date = jb - jd - (int) (30.6001 * je);
    int month = je - 1;
    if (month > 12) month -= 12;
    int year = jc - 4715;
    if (month > 2) --year;
    if (year <= 0) --year;
    return new int[] { year, month, date }; //continues
Third Implementation

190:    private int year;
191:    private int month;
192:    private int date;
193:    private int julian;
194:    private boolean ymdValid;
195:    private boolean julianValid;
196:    }
01: public class DayTester
02: {
03:   public static void main(String[] args)
04:   {
05:     Day today = new Day(2001, 2, 3); // February 3, 2001
06:     Day later = today.addDays(999);
07:     System.out.println(later.getYear() + "-" + later.getMonth() + "-" + later.getDate());
08:     System.out.println(later.daysFrom(today)); // Prints 999
09:   }
10: }
11: }
12: }
Comparison of implementations

• Which implementation is best?
  • No single answer – depends on usage
• 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
  • Can't just use text editor to replace all \texttt{d.year} with \texttt{d.getYear()}
    • \texttt{d.year++}?
  • \texttt{d = new Day(d.getDay(), d.getMonth(), d.getYear() + 1)}
  • Ugh--that gets really inefficient in Julian representation
  • Don't use public fields, even for "simple" classes
Accessors and Mutators

- Mutator: Changes object state
- Accessor: Reads object state without changing it
- Day class has no mutators!
- Class without mutators is immutable
- String is immutable
- Date and GregorianCalendar are mutable

Don't Supply a Mutator for every Accessor
- Day has `getYear`, `getMonth`, `getDate` accessors
- Day does not have `setYear`, `setMonth`, `setDate` mutators
- These mutators would not work well

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

- Maybe we should call `setDate` first?
  ```
  Day deadline = new Day(2001, 2, 28);
deadline.setDate(31); // ERROR
deadline.setMonth(3);
  ```
- GregorianCalendar implements confusing rollover.
- Silently gets the wrong result instead of error.
- Immutability is useful
Sharing Mutable References

• References to immutable objects can be freely shared
• Don't share mutable references
• Example

```java
class Employee {
    private String name;
    private double salary;
    private Date hireDate;
    
    public String getName() { return name; }
    public double getSalary() { return salary; }
    public Date getHireDate() { return hireDate; }
}
```
Pitfall:

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

Remedy: Use clone

public Date getHireDate()
{
    return (Date)hireDate.clone();
}
Sharing Mutable References

```
harry = Employee
    name = "harry"
    hireDate = 

d = Date
```
Final Instance Fields
• Good idea to mark immutable instance fields as final
  • private final int day;
  • final object reference can still refer to mutating object
  • 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:
  ```java
  Scanner in = . . .;
  String s = in.next();
  ```
  - Yields current token and advances iteration.
- What if I want to read the current token again?
Separating Accessors and Mutators

• Better interface:
  
  ```java
  String getCurrent();
  void next();
  ```

• Even more convenient:
  
  ```java
  String getCurrent();
  String next(); // 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

• Side effect of a method: any observable state change
• Mutator: changes implicit parameter
• Other side effects:
  • change to explicit parameter
  • change to accessible static object
• Avoid these side effects--they confuse users
• Good example, no side effect beyond implicit parameter
  • a.addAll(b)
  • mutates a but not b
Side Effects

Date formatting (basic):
SimpleDateFormat formatter = . . .;
String dateString = "January 11, 2012";
Date d = formatter.parse(dateString);

Advanced:
FieldPosition position = . . .;
Date d = formatter.parse(dateString, position);

Formats the date string within a larger string, starting at position, after which the position is updated to the character following the date.

Side effect: updates position parameter

Design could be better: add position to formatter state

However, that limits the usage of a SimpleDateFormat object.
Side Effects

- Avoid modifying static objects
- Example: System.out
- Don't print error messages to System.out:
  - if (newMessages.isFull())
  - System.out.println("Sorry--no space");
- Your classes may need to run in an environment without System.out
- Rule of thumb: Minimize side effects beyond implicit parameter
The Law of Demeter

• Example: Mail system in chapter 2
• Mailbox currentMailbox = mailSystem.findMailbox(...);
• Breaks encapsulation in a subtle way – Connection object can change a field of a Mailbox object that is sent to it.
• Suppose future version of MailSystem uses a database
• Then it no longer has mailbox objects
• Common in larger systems
• Karl Lieberherr: Law of Demeter (demEeter)
• Demeter = Greek goddess of agriculture, sister of Zeus
The Law of Demeter

• The law: 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
• Remedy in mail system: Delegate mailbox methods to mail system
  • mailSystem.getCurrentMessage(int mailboxNumber);
  • mailSystem.addMessage(int mailboxNumber, Message msg);
  • 
• Rule of thumb, not a mathematical law
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
• Bad example:

```java
public class Mailbox
{
    public addMessage(Message aMessage) { ... }
    public Message getCurrentMessage() { ... }
    public Message removeCurrentMessage() { ... }
    public void processCommand(String command) { ... }
    ...
}
```

• What is the purpose of `processCommand()`?
• Creates a difficult system to understand and develop.
Completeness

• Support operations that are well-defined on abstraction

• Potentially bad example: Date
  
  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?
• After all, we have before, after, getTime
Convenience

• A good interface makes all tasks possible . . . and common tasks simple
• Bad example: Reading from System.in before Java 5.0

    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

• Why doesn't System.in have a readLine method?
• After all, System.out has println.
• Scanner class fixes inconvenience
Clarity

• Confused programmers write buggy code
• Bad example: Removing elements from LinkedList
• Reminder: Standard linked list class
  ```java
  LinkedList<String> countries = new LinkedList<String>();
countries.add("A");
countries.add("B");
countries.add("C");
  ```
• Iterate through list:
  ```java
  ListIterator<String> iterator = countries.listIterator();
  while (iterator.hasNext())
    System.out.println(iterator.next());
  ```
Clarity

- Iterator between elements
- Like blinking caret in word processor
- add adds to the left of iterator (like word processor):
  - Add X before B:
    ```java
    ListIterator<String> iterator = countries.listIterator(); // |ABC
    iterator.next(); // A|BC
    iterator.add("France"); // AX|BC
    ```
- To remove first two elements, you can't just "backspace"
- 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:
  ```java
  new GregorianCalendar(year, month - 1, day)
  ```
- Why is month 0-based?
Consistency

- Bad example: String class has:
  - `s.equals(t)`
  - `s.equalsIgnoreCase(t)`
- But
  ```java
  boolean regionMatches(int toffset,
                         String other, int ooffset, int len)
  boolean regionMatches(boolean ignoreCase, int toffset,
                         String other, int ooffset, int len)
  ```
- Why not `regionMatchesIgnoreCase`?
- Very common problem in student code
Programming by Contract

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

• Caller attempts to remove a message from an empty MessageQueue

• What should happen?
  • MessageQueue can declare this as an error.
  • MessageQueue can tolerate the call and return a dummy value such as null.

• Which is better?
  • Excessive error checking is costly
  • Returning dummy values can complicate testing
Preconditions

• Contract metaphor
  • Service provider must specify preconditions
  • If precondition is fulfilled, service provider must work correctly
  • Otherwise, service provider can do anything

• When precondition fails, service provider may
  • throw exception
  • return false answer
  • corrupt data
Preconditions

/**
 * Remove message at head
 * @return the message at the head
 * @precondition size() > 0
 */
Message remove()
{
    return elements.remove(0);
}

• What happens if precondition not fulfilled?
  • IndexOutOfBoundsException
  • Other implementation may have different behavior
• Note: For @precondition to show up in JavaDoc, run javadoc with the option
  -tag precondition:cm "Precondition:"
• The cm instructs javadoc to look for this tag only in constructors and methods.
Circular Array Implementation

• Efficient implementation of bounded queue
• Avoids inefficient shifting of elements
• Circular: head, tail indexes wrap around
• Inefficient removal of an element:
A circular array with ‘wrap around’
A first-in, first-out bounded collection of messages.

public class MessageQueue {
    Constructs an empty message queue.  
    @param capacity the maximum capacity of the queue 
    @precondition capacity > 0 
    */
    public MessageQueue(int capacity) {
        elements = new Message[capacity];
        count = 0;
        head = 0;
        tail = 0;
    }
Message Queue

19:    /**
20:        Remove message at head.
21:        @return the message that has been removed from the queue
22:        @precondition size() > 0
23:     */
24:    public Message remove()
25:    {  
26:       Message r = elements[head];
27:       head = (head + 1) % elements.length;
28:       count--;
29:       return r;
30:    }
/**
 * Append a message at tail.
 * @param aMessage the message to be appended
 * @precondition !isFull();
 */
public void add(Message aMessage)
{
elements[tail] = aMessage;
tail = (tail + 1) % elements.length;
count++;
}

/**
 * Get the total number of messages in the queue.
 * @return the total number of messages in the queue
 */
public int size()
{
    return count;
}
/**
 * Checks whether this queue is full
 * @return true if the queue is full
 */
public boolean isFull()
{
    return count == elements.length;
}
/**
 * Get message at head.
 * @return the message that is at the head of the queue
 * @precondition size() > 0
 */
public Message peek()
{
    return elements[head];
}
private Message[] elements;
private int head;
private int tail;
private int count;
}
Preconditions

• In circular array implementation, failure of remove precondition corrupts queue!
• Bounded queue needs precondition for add
• Naive approach:
  @precondition size() < elements.length
• Precondition should be checkable by caller
• Better:
  @precondition size() < getCapacity()
Assertions

• Mechanism for warning programmers
• Can be turned off after testing
• Useful for warning programmers about precondition failure
• Syntax:
  assert condition;
  assert condition : explanation;
• Throws AssertionError if condition false and checking enabled
public Message remove()
{
    assert count > 0 : "violated precondition size() > 0";
    Message r = elements[head];
    ... 
}

• During testing, run with
  java -enableassertions MyProg
• Or shorter, java -ea
Exceptions in the Contract

/**
 ...  
@throws NoSuchElementException if queue is empty
*/

public Message remove()
{
  if (count == 0)
    throw new NoSuchElementException();
  Message r = elements[head];
  ...
}

• Exception throw part of the contract
• Caller can rely on behavior
• Exception throw not result of precondition violation
• This method has no precondition
Postconditions

• Conditions that the service provider guarantees
• Every method promises description, @return
• Sometimes, can assert additional useful condition
• Example: add method
  
  @postcondition size() > 0
• Postcondition of one call can imply precondition of another:
  
  q.add(m1);
  m2 = q.remove();
Class Invariants

• Condition that is true after every constructor
• Preserved by every method
• (if it's true before the call, it's again true afterwards)
• Useful for checking validity of operations
Class Invariants

- Example: Circular array queue
  \[0 \leq \text{head} \land \text{head} < \text{elements.length}\]
- First check that it's true for constructor
  - Sets head = 0
  - Need precondition size > 0!
- Check mutators. Start with remove
  - Sets \(\text{head}_{\text{new}} = (\text{head}_{\text{old}} + 1) \mod \text{elements.length}\)
  - We know \(\text{head}_{\text{old}} \geq 0\) (Why?)
  - \% operator property:
    - \(0 \leq \text{head}_{\text{new}} \land \text{head}_{\text{new}} < \text{elements.length}\)
  - What's the use? Array accesses are correct!
- You can similarly prove that:
  - \(0 \leq \text{tail} \land \text{tail} < \text{elements.length}\)
  - is an invariant
Unit Testing

- Unit test = test of a single class
- Design test cases during implementation
- Run tests after every implementation change
- When you find a bug, add a test case that catches it
JUnit

- Convention: Test class name = tested class name + Test
- Test methods’ names start with test

```java
import junit.framework.*;
public class DayTest extends TestCase {

    public void testAdd() { ... }
    public void testDaysBetween() { ... }
    ...
}
```
JUnit

- Each test case ends with `assertTrue` method
- (or another JUnit assertion method such as `assertEquals`)
- Test framework catches assertion failures

```java
public void testAdd()
{
    Day d1 = new Day(1970, 1, 1);
    int n = 1000;
    Day d2 = d1.addDays(n);
    assertTrue(d2.daysFrom(d1) == n);
}
```
End