

---

# COMP 303 - Lecture Notes for Week 3 - Class Design and JUnit testing

- Slides edited from, Object-Oriented Design & Patterns, by Cay S. Horstmann
- Original slides available from:  
[http://www.horstmann.com/design\\_and\\_patterns.html](http://www.horstmann.com/design_and_patterns.html)
- Modifications made by Laurie Hendren, McGill University
- Topics this week:
  - Class Design, Chapter 3
  - Using JUnit for unit testing

---

next .... [Slide 1] ....

---

## 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

---

[previous](#) | [start](#) | [next](#) ... [Slide 2] ....

---

## Date Classes in Standard Library

- 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*

---

[previous](#) | [start](#) | [next](#) ... [Slide 3] ....

---

## Methods of the Date class

<code>boolean after(Date other)</code>	Tests if this date is after the specified date
<code>boolean before(Date other)</code>	Tests if this date is before the specified date
<code>int compareTo(Date other)</code>	Tells which date came before the other
<code>long getTime()</code>	Returns milliseconds since the epoch (1970-01-01 00:00:00 GMT)
<code>void setTime(long n)</code>	Sets the date to the given number of milliseconds since the epoch

---

[previous](#) | [start](#) | [next](#) ... [Slide 4] ...

---

## Methods of the Date class

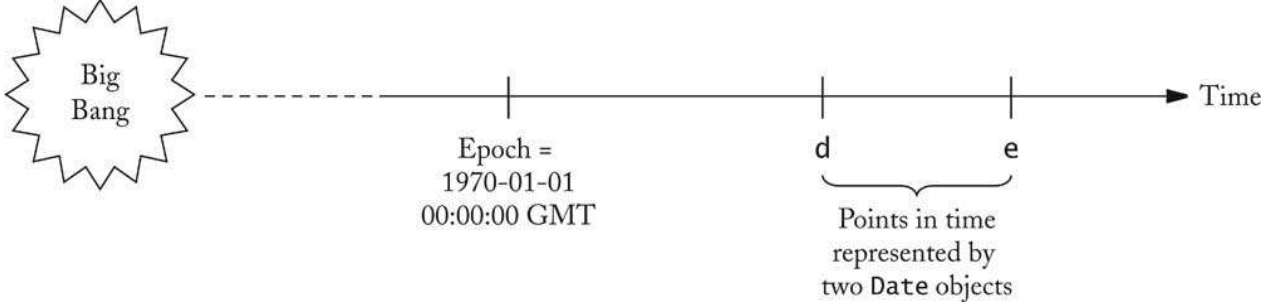
- Deprecated methods omitted
- Date class methods supply *total ordering* on Date objects
- Convert to scalar time measure
- Note that `before/after` not strictly necessary  
(Presumably introduced for convenience)

---

[previous](#) | [start](#) | [next](#) ... [Slide 5] ...

---

# Points in Time



---

[previous](#) | [start](#) | [next](#) ... [Slide 6] ...

---

# The GregorianCalendar Class

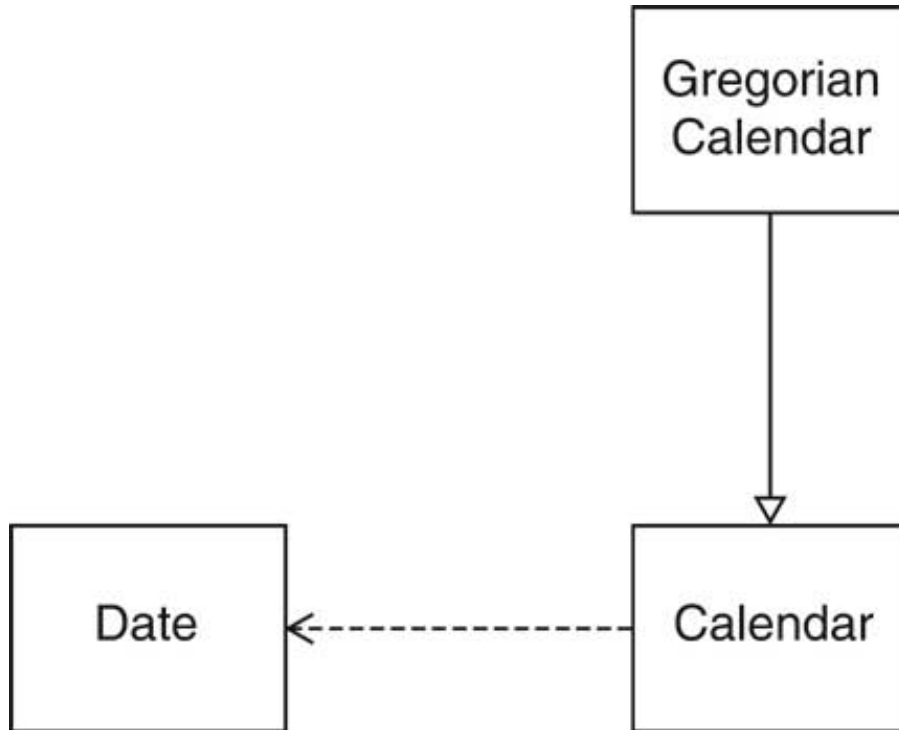
- The Date class doesn't measure months, weekdays, etc.
- That's the job of a *calendar*
- A calendar assigns a name to a point in time
- Many calendars in use:
  - 
  - Gregorian
  - Contemporary: Hebrew, Arabic, Chinese
  - Historical: French Revolutionary, Mayan

---

[previous](#) | [start](#) | [next](#) ... [Slide 7] ....

---

## Date Handling in the Java Library





---

## 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
- Answer questions such as:
  - How many days are there between now and the end of the year?
  - What day is 100 days from now?

---

[previous](#) | [start](#) | [next](#) ... [Slide 9] ...

---

# Designing a Day Class

Day
<i>relate calendar days to day counts</i>

---

## Designing a Day Class

- `daysFrom` computes number of days between two days:

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

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

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

- **Mathematical relationship:**

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

- Clearer when written with "overloaded operators":

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

- Constructor `Date(int year, int month, int date)`
- `getYear`, `getMonth`, `getDate` accessors

---

# Implementing a Day Class

- Straightforward implementation:

```
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
- javadoc of API of Day (using -public flag)

Package [Class](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

[SUMMARY: NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[DETAIL: FIELD](#) | [CONSTR](#) | [METHOD](#)

---

## Class Day

java.lang.Object

└ **Day**

---

public class **Day**

extends java.lang.Object

---

### Constructor Summary

**Day**(int aYear, int aMonth, int aDate)

Constructs a day with a given year, month, and day of the Julian/Gregorian calendar.

## Method Summary

<code>Day</code>	<b><code>addDays</code></b> (int n) Returns a day that is a certain number of days away from this day
int	<b><code>daysFrom</code></b> ( <code>Day</code> other) Returns the number of days between this day and another day
int	<b><code>getDate</code></b> () Returns the day of the month of this day
int	<b><code>getMonth</code></b> () Returns the month of this day
int	<b><code>getYear</code></b> () Returns the year of this day

## Methods inherited from class `java.lang.Object`

`equals`, `getClass`, `hashCode`, `notify`, `notifyAll`,  
`toString`, `wait`, `wait`, `wait`

## Constructor Detail

# Day

```
public Day(int aYear,  
           int aMonth,  
           int aDate)
```

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

## Parameters:

aYear - a year != 0

aMonth - a month between 1 and 12

aDate - a date between 1 and 31

## Method Detail

### getYear

```
public int getYear()
```

Returns the year of this day

#### Returns:

the year

---

### getMonth

```
public int getMonth()
```

Returns the month of this day

**Returns:**

the month

---

**getDate**

```
public int getDate()
```

Returns the day of the month of this day

**Returns:**

the day of the month

---

**addDays**

```
public Day addDays(int n)
```

Returns a day that is a certain number of days away from this day

**Parameters:**

n - the number of days, can be negative

**Returns:**

a day that is n days away from this one

---

**daysFrom**

```
public int daysFrom(Day other)
```

Returns the number of days between this day and another day

**Parameters:**

other - the other day



**Returns:**

the number of days that this day is away from the other (>0 if this day comes later)

---

Package **Class [Tree](#) [Deprecated](#) [Index](#) [Help](#)**

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

---

---

## Implementing a Day Class

- [Ch3/code/day1/Day.java](#)
- [Ch3/code/day1/DayTest.java](#)
- Note private helper methods
- Computations are inefficient: a day at a time

---

[previous](#) | [start](#) | [next](#) ... [Slide 13] ...

```
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:     }
017:
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:
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:
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:
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: }
090:
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:
107: /**
108:     Computes the next day.
109:     @return the day following this day
110: */
111: private Day nextDay()
112: {
113:     int y = year;
114:     int m = month;
115:     int d = date;
116:
117:     if (y == GREGORIAN_START_YEAR
118:         && m == GREGORIAN_START_MONTH
119:         && d == JULIAN_END_DAY)
120:         d = GREGORIAN_START_DAY;
121:     else if (d < daysPerMonth(y, m))
122:         d++;
123:     else

```

```

124:     {
125:         d = 1;
126:         m++;
127:         if (m > DECEMBER)
128:         {
129:             m = JANUARY;
130:             y++;
131:             if (y == 0) y++;
132:         }
133:     }
134:     return new Day(y, m, d);
135: }
136:
137: /**
138:     Computes the previous day.
139:     @return the day preceding this day
140: */
141: private Day previousDay()
142: {
143:     int y = year;
144:     int m = month;
145:     int d = date;
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:     {
155:         m--;
156:         if (m < JANUARY)
157:         {
158:             m = DECEMBER;
159:             y--;
160:             if (y == 0) y--;
161:         }
162:         d = daysPerMonth(y, m);
163:     }
164:     return new Day(y, m, d);

```

```

165:     }
166:
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:
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:     {
188:         if (y % 4 != 0) return false;
189:         if (y < GREGORIAN_START_YEAR) return true;
190:         return (y % 100 != 0) || (y % 400 == 0);
191:     }
192:
193:     private int year;
194:     private int month;
195:     private int date;
196:
197:     private static final int[] DAYS_PER_MONTH
198:         = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
199:
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;
204:
205:     public static final int JANUARY = 1;

```

```
206:    public static final int FEBRUARY = 2;
207:    public static final int DECEMBER = 12;
208: }
209:
210:
211:
212:
213:
```



```
01: public class DayTest
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()
08:             + "-" + later.getMonth()
09:             + "-" + later.getDate());
10:         System.out.println(later.daysFrom(today)); // prints 999
11:     }
12: }
```

---

## Second Implementation

- 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
- [Ch3/code/day2/Day.java](#)

---

[previous](#) | [start](#) | [next](#) .... [Slide 14] ....

```
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:         julian = toJulian(aYear, aMonth, aDate);
014:     }
015:
016:     /**
017:         Returns the year of this day
018:         @return the year
019:     */
020:     public int getYear()
021:     {
022:         return fromJulian(julian)[0];
023:     }
024:
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:
043:  /**
044:     Returns a day that is a certain number of days away from
045:     this day
046:     @param n the number of days, can be negative
047:     @return a day that is n days away from this one
048:  */
049:  public Day addDays(int n)
050:  {
051:      return new Day(julian + n);
052:  }
053:
054:  /**
055:     Returns the number of days between this day and another day.
056:     @param other the other day
057:     @return the number of days that this day is away from
058:     the other (>0 if this day comes later)
059:  */
060:  public int daysFrom(Day other)
061:  {
062:      return julian - other.julian;
063:  }
064:
065:  private Day(int aJulian)
066:  {
067:      julian = aJulian;
068:  }
069:
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 date
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
086:      in C, 2nd ed., Cambridge University Press 1992
087:      */
088:      private static int toJulian(int year, int month, int date)
089:      {
090:          int jy = year;
091:          if (year < 0) jy++;
092:          int jm = month;
093:          if (month > 2) jm++;
094:          else
095:          {
096:              jy--;
097:              jm += 13;
098:          }
099:          int jul = (int) (java.lang.Math.floor(365.25 * jy)
100:              + java.lang.Math.floor(30.6001*jm) + date + 1720995.0);
101:
102:          int IGREG = 15 + 31*(10+12*1582);
103:              // Gregorian Calendar adopted Oct. 15, 1582
104:
105:          if (date + 31 * (month + 12 * year) >= IGREG)
106:              // change over to Gregorian calendar
107:          {
108:              int ja = (int)(0.01 * jy);
109:              jul += 2 - ja + (int)(0.25 * ja);
110:          }
111:          return jul;
112:      }
113:
114:      /**
115:      Converts a Julian day number to a calendar date.
116:
117:      This algorithm is from Press et al., Numerical Recipes
118:      in C, 2nd ed., Cambridge University Press 1992
119:
120:      @param j the Julian day number
121:      @return an array whose 0 entry is the year, 1 the month,
122:      and 2 the day of the month.
123:      */

```

```

124:     private static int[] fromJulian(int j)
125:     {
126:         int ja = j;
127:
128:         int JGREG = 2299161;
129:         /* the Julian day number of the adoption of the Gregorian
130:           calendar
131:           */
132:
133:         if (j >= JGREG)
134:           /* cross-over to Gregorian Calendar produces this
135:             correction
136:             */
137:         {
138:             int jalpha = (int)((float)(j - 1867216) - 0.25)
139:               / 36524.25);
140:             ja += 1 + jalpha - (int)(0.25 * jalpha);
141:         }
142:         int jb = ja + 1524;
143:         int jc = (int)(6680.0 + ((float)(jb-2439870) - 122.1)
144:           /365.25);
145:         int jd = (int)(365 * jc + (0.25 * jc));
146:         int je = (int)((jb - jd)/30.6001);
147:         int date = jb - jd - (int)(30.6001 * je);
148:         int month = je - 1;
149:         if (month > 12) month -= 12;
150:         int year = jc - 4715;
151:         if (month > 2) --year;
152:         if (year <= 0) --year;
153:         return new int[] { year, month, date };
154:     }
155:
156:     private int julian;
157: }
158:
159:
160:
161:
162:

```

---

## Third Implementation

- Now constructor, accessors are inefficient
- Best of both worlds: Cache known Julian, y/m/d values
- [Ch3/code/day3/Day.java](#)
- Which implementation is best?

---

[previous](#) | [start](#) | [next](#) .... [Slide 15] ....

```
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:         ymdValid = true;
017:         julianValid = false;
018:     }
019:
020:     /**
021:         Returns the year of this day
022:         @return the year
023:     */
024:     public int getYear()
025:     {
026:         ensureYmd();
027:         return year;
028:     }
029:
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:     }
049:
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:     {
058:         ensureJulian();
059:         return new Day(julian + n);
060:     }
061:
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:     }
075:
076:     private Day(int aJulian)
077:     {
078:         julian = aJulian;
079:         ymdValid = false;
080:         julianValid = true;
081:     }
082:

```

```

083:  /**
084:      Computes the Julian day number of this day if
085:      necessary
086:      */
087:
088:  private void ensureJulian()
089:  {
090:      if (julianValid) return;
091:      julian = toJulian(year, month, date);
092:      julianValid = true;
093:  }
094:
095:  /**
096:      Converts this Julian day number to a calendar date if necessary
097:      */
098:
099:  private void ensureYmd()
100:  {
101:      if (ymdValid) return;
102:      int[] ymd = fromJulian(julian);
103:      year = ymd[0];
104:      month = ymd[1];
105:      date = ymd[2];
106:      ymdValid = true;
107:  }
108:
109:  /**
110:      Computes the Julian day number of the given day.
111:
112:      @param year a year
113:      @param month a month
114:      @param date a date
115:      @return The Julian day number that begins at noon of
116:      the given day
117:      Positive year signifies CE, negative year BCE.
118:      Remember that the year after 1 BCE is 1 CE.
119:
120:      A convenient reference point is that May 23, 1968 noon
121:      is Julian day number 2440000.
122:
123:      Julian day number 0 is a Monday.

```

```

124:
125:     This algorithm is from Press et al., Numerical Recipes
126:     in C, 2nd ed., Cambridge University Press 1992
127:     */
128:     private static int toJulian(int year, int month, int date)
129:     {
130:         int jy = year;
131:         if (year < 0) jy++;
132:         int jm = month;
133:         if (month > 2) jm++;
134:         else
135:         {
136:             jy--;
137:             jm += 13;
138:         }
139:         int jul = (int) (java.lang.Math.floor(365.25 * jy)
140:             + java.lang.Math.floor(30.6001*jm) + date + 1720995.0);
141:
142:         int IGREG = 15 + 31*(10+12*1582);
143:         // Gregorian Calendar adopted Oct. 15, 1582
144:
145:         if (date + 31 * (month + 12 * year) >= IGREG)
146:             // change over to Gregorian calendar
147:         {
148:             int ja = (int)(0.01 * jy);
149:             jul += 2 - ja + (int)(0.25 * ja);
150:         }
151:         return jul;
152:     }
153:
154:     /**
155:         Converts a Julian day number to a calendar date.
156:
157:         This algorithm is from Press et al., Numerical Recipes
158:         in C, 2nd ed., Cambridge University Press 1992
159:
160:         @param j  the Julian day number
161:         @return an array whose 0 entry is the year, 1 the month,
162:         and 2 the day of the month.
163:     */
164:     private static int[] fromJulian(int j)

```

```

165:     {
166:         int ja = j;
167:
168:         int JGREG = 2299161;
169:         /* the Julian day number of the adoption of the Gregorian
170:            calendar
171:            */
172:
173:         if (j >= JGREG)
174:         /* cross-over to Gregorian Calendar produces this
175:            correction
176:            */
177:         {
178:             int jalpha = (int)(((float)(j - 1867216) - 0.25)
179:                / 36524.25);
180:             ja += 1 + jalpha - (int)(0.25 * jalpha);
181:         }
182:         int jb = ja + 1524;
183:         int jc = (int)(6680.0 + ((float)(jb-2439870) - 122.1)
184:            /365.25);
185:         int jd = (int)(365 * jc + (0.25 * jc));
186:         int je = (int)((jb - jd)/30.6001);
187:         int date = jb - jd - (int)(30.6001 * je);
188:         int month = je - 1;
189:         if (month > 12) month -= 12;
190:         int year = jc - 4715;
191:         if (month > 2) --year;
192:         if (year <= 0) --year;
193:         return new int[] { year, month, date };
194:     }
195:
196:     private int year;
197:     private int month;
198:     private int date;
199:     private int julian;
200:     private boolean ymdValid;
201:     private boolean julianValid;
202: }

```

---

# 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

```
d.year
```

with

```
d.getYear()
```

- How about

```
d.year++?
```

- 

```
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

---

[previous](#) | [start](#) | [next](#) .... [Slide 17] ....

---

## 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

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



---

# Sharing Mutable References

- Pitfall:

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

- Remedy: Use clone

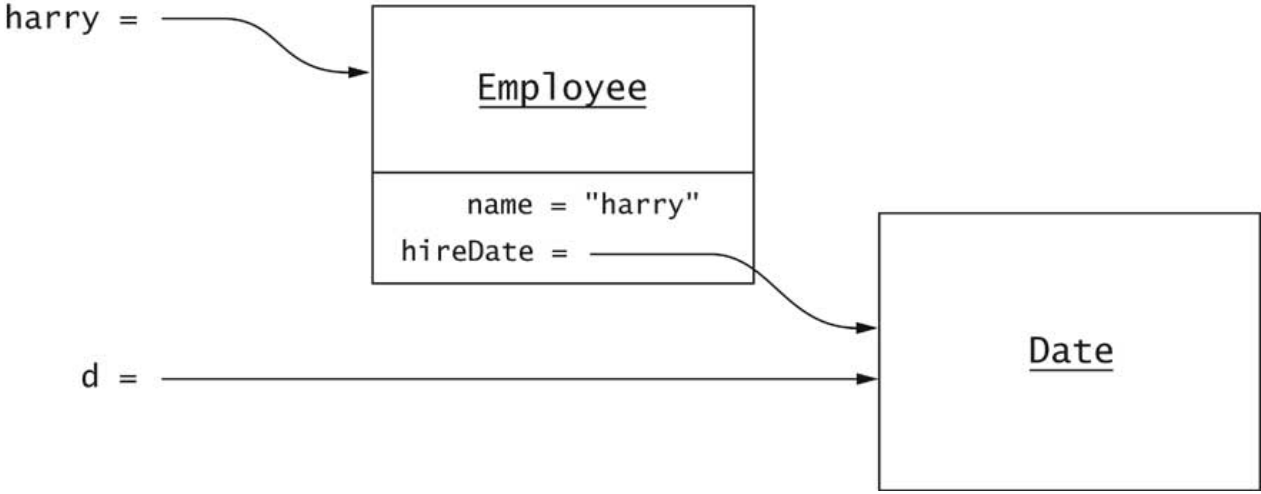
```
public Date getHireDate()  
{  
    return (Date)hireDate.clone();  
}
```

---

[previous](#) | [start](#) | [next](#) .... [Slide 20] ....

---

# Sharing Mutable References



---

## 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

---

[previous](#) | [start](#) | [next](#) .... [Slide 22] ....

---

## 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:

```
StringTokenizer t = . . . ;  
String s = t.nextToken();
```

- Yields current token *and* advances iteration
- What if I want to read the current token again?

---

[previous](#) | [start](#) | [next](#) .... [Slide 23] ....

---

## Separating Accessors and Mutators

- Better interface:

```
String getToken();  
void nextToken();
```

- Even more convenient:

```
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

---

[previous](#) | [start](#) | [next](#) .... [Slide 24] ....

---

## 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
- Good example, no side effect:

```
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);
```

- Side effect: updates `position`
- Design could be better: add `position` to formatter state

---

## 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

---

[previous](#) | [start](#) | [next](#) .... [Slide 27] ....



---

## Law of Demeter

- Example: Mail system in chapter 2 `Mailbox` `currentMailbox = mailSystem.findMailbox(...);`
- Breaks encapsulation
- Suppose future version of `MailSystem` uses a database
- Then it no longer has mailbox objects
- Common in larger systems
- Karl Lieberherr: Law of Demeter
- Demeter = Greek goddess of agriculture, sister of Zeus

---

[previous](#) | [start](#) | [next](#) .... [Slide 28] ....

---

## 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

---

[previous](#) | [start](#) | [next](#) .... [Slide 29] ....

---

## Quality of Class Interface

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

---

[previous](#) | [start](#) | [next](#) .... [Slide 30] ....

---

# Cohesion

- Class describes a *single* abstraction
- Methods should be related to the single abstraction
- Bad example:

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

---

[previous](#) | [start](#) | [next](#) .... [Slide 31] ....

---

# 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`

---

[previous](#) | [start](#) | [next](#) .... [Slide 32] ....

---

## 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?
- After all, `System.out` has `println`.
- Why can't I make a `BufferedReader` from an input stream?
- After all, I can make a `PrintWriter` from an output stream.

---

[previous](#) | [start](#) | [next](#) .... [Slide 33] ....

---

# Clarity

- Confused programmers write buggy code
- Bad example: Removing elements from `LinkedList`
- Reminder: Standard linked list class

```
LinkedList countries = new LinkedList();  
countries.add("A");  
countries.add("B");  
countries.add("C");
```

- Iterate through list:

```
ListIterator iterator = countries.listIterator();  
while (iterator.hasNext())  
    System.out.println(iterator.next());
```

---

[previous](#) | [start](#) | [next](#) .... [Slide 34] ....

---

# Clarity

- Iterator *between* elements
- Like blinking caret in word processor
- add adds to the left of iterator (like word processor):
- Add X before B:

```
ListIterator 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?

---

[previous](#) | [start](#) | [next](#) .... [Slide 35] ....



---

# 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?

---

[previous](#) | [start](#) | [next](#) .... [Slide 36] ....

---

# Consistency

- Bad example: String class

```
s.equals(t) / s.equalsIgnoreCase(t)
```

- But

```
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

---

[previous](#) | [start](#) | [next](#) .... [Slide 37] ....

---

# Programming by Contract

- Spell out responsibilities
- - of caller
  - of implementor
- Increase reliability
- Increase efficiency

---

[previous](#) | [start](#) | [next](#) .... [Slide 38] ....

---

# Preconditions

- Caller attempts to remove message from empty MessageQueue
- What should happen?
- MessageQueue can declare this as an error
- MessageQueue can tolerate call and return dummy value
- What is better?

---

[previous](#) | [start](#) | [next](#) ... [Slide 39] ...

---

# 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*
- 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 removeFirst()  
{  
    return (Message)elements.remove(0);  
}
```

- What happens if precondition not fulfilled?
- `IndexOutOfBoundsException`
- Other implementation may have different behavior

---

[previous](#) | [start](#) | [next](#) .... [Slide 41] ....

---

# Circular Array Implementation

- Efficient implementation of bounded queue
- Avoids inefficient shifting of elements
- Circular: head, tail indexes wrap around
- [Ch3/queue/MessageQueue.java](#)

---

[previous](#) | [start](#) | [next](#) .... [Slide 42] ....

```

01: /**
02:     A first-in, first-out bounded collection of messages.
03: */
04: public class MessageQueue
05: {
06:     /**
07:         Constructs an empty message queue.
08:         @param capacity the maximum capacity of the queue
09:         @precondition capacity > 0
10:     */
11:     public MessageQueue(int capacity)
12:     {
13:         elements = new Message[capacity];
14:         count = 0;
15:         head = 0;
16:         tail = 0;
17:     }
18:
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 removeFirst()
25:     {
26:         Message r = elements[head];
27:         head = (head + 1) % elements.length;
28:         count--;
29:         return r;
30:     }
31:
32:     /**
33:         Append a message at tail.
34:         @param aMessage the message to be appended
35:         @precondition !isFull();
36:     */
37:     public void add(Message aMessage)
38:     {
39:         elements[tail] = aMessage;
40:         tail = (tail + 1) % elements.length;
41:         count++;

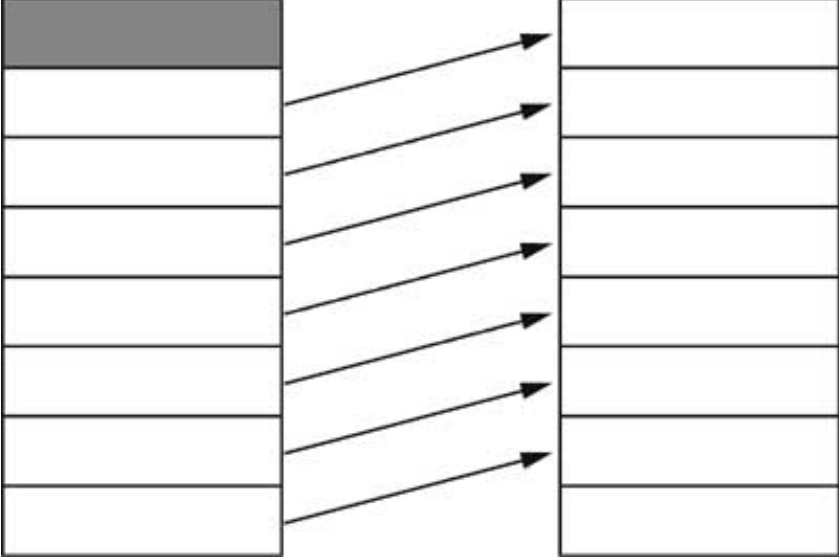
```



```
42:     }
43:
44:     /**
45:         Get the total number of messages in the queue.
46:         @return the total number of messages in the queue
47:     */
48:     public int size()
49:     {
50:         return count;
51:     }
52:
53:     /**
54:         Checks whether this queue is full
55:         @return true if the queue is full
56:     */
57:     public boolean isFull()
58:     {
59:         return count == elements.length;
60:     }
61:
62:     /**
63:         Get message at head.
64:         @return the message that is at the head of the queue
65:         @precondition size() > 0
66:     */
67:     public Message getFirst()
68:     {
69:         return elements[head];
70:     }
71:
72:     private Message[] elements;
73:     private int head;
74:     private int tail;
75:     private int count;
76: }
```

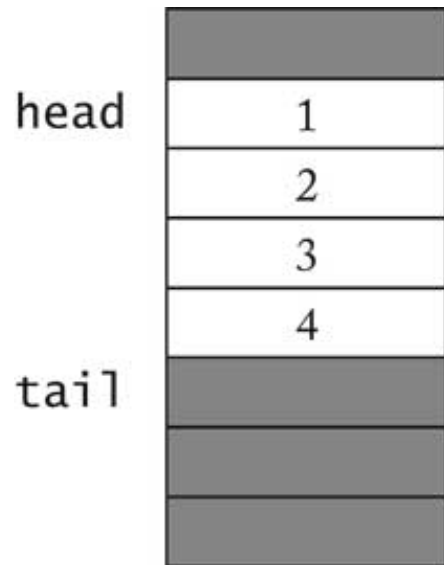
---

# Inefficient Shifting of Elements



---

## A Circular Array

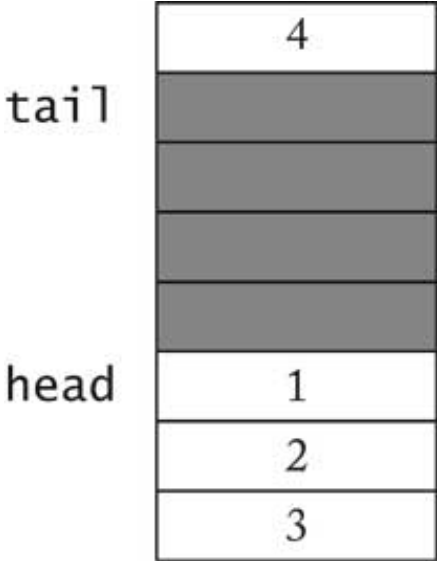


---

[previous](#) | [start](#) | [next](#) .... [Slide 44] ....

---

# Wrapping around the End



---

# 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()`

---

[previous](#) | [start](#) | [next](#) .... [Slide 46] ....

---

# 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

---

[previous](#) | [start](#) | [next](#) .... [Slide 47] ....

---

# Assertions

```
public Message removeFirst()
{
    assert count > 0 : "violated precondition size() > 0";
    Message r = elements[head];
    . . .
}
```

- During testing, run with

```
java -enableassertions MyProg
```

---

[previous](#) | [start](#) | [next](#) .... [Slide 48] ....

---

# Exceptions in the Contract

```
/**
 * . . .
 * @throws IllegalArgumentException if queue is empty
 */
public Message removeFirst()
{
    if (count == 0)
        throw new IllegalArgumentException();
    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();
```

---

[previous](#) | [start](#) | [next](#) .... [Slide 50] ....

---

# 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

---

[previous](#) | [start](#) | [next](#) .... [Slide 51] ....

---

# Class Invariants

- Example: Circular array queue

```
0 <= head && head < elements.length
```

- First check it's true for constructor

- 

- Sets `head = 0`

- Need precondition `size > 0!`

- Check mutators. Start with `add`

- 

- Sets `headnew = (headold + 1) % elements.length`

- We know `headold > 0` (Why?)

- `%` operator property:

```
0 <= headnew && headnew < elements.length
```

- What's the use? Array accesses are correct!

```
return elements[head];
```

---

[previous](#) | [start](#) | [next](#) .... [Slide 52] ....

---

## Class Invariants

- Example: Queue with array list  
elements only contains objects of type Message
- Ch2/mail/MessageQueue.java
- It's true for constructor.  
elements is initially empty
- Check mutators. Start with add  
Parameter type is Message
- What's the use? Casts are correct!  
`return (Message)elements.remove(0);`

---

[previous](#) | [start](#) | [next](#) .... [Slide 53] ....

```
01: import java.util.ArrayList;
02:
03: /**
04:     A first-in, first-out collection of messages. This
05:     implementation is not very efficient. We will consider
06:     a more efficient implementation in chapter 3.
07: */
08: public class MessageQueue
09: {
10:     /**
11:         Constructs an empty message queue.
12:     */
13:     public MessageQueue()
14:     {
15:         queue = new ArrayList();
16:     }
17:
18:     /**
19:         Remove message at head.
20:         @return message that has been removed from the queue
21:     */
22:     public Message removeFirst()
23:     {
24:         return (Message) queue.remove(0);
25:     }
26:
27:     /**
28:         Append message at tail.
29:         @param newMessage the message to be appended
30:     */
31:     public void add(Message newMessage)
32:     {
33:         queue.add(newMessage);
34:     }
35:
36:     /**
37:         Get the total number of messages in the queue.
38:         @return the total number of messages in the queue
39:     */
40:     public int size()
41:     {
```

```
42:         return queue.size();
43:     }
44:
45:     /**
46:         Get message at head.
47:         @return message that is at the head of the queue
48:     */
49:     public Message getFirst()
50:     {
51:         return (Message) queue.get(0);
52:     }
53:
54:     private ArrayList queue;
55: }
```

---

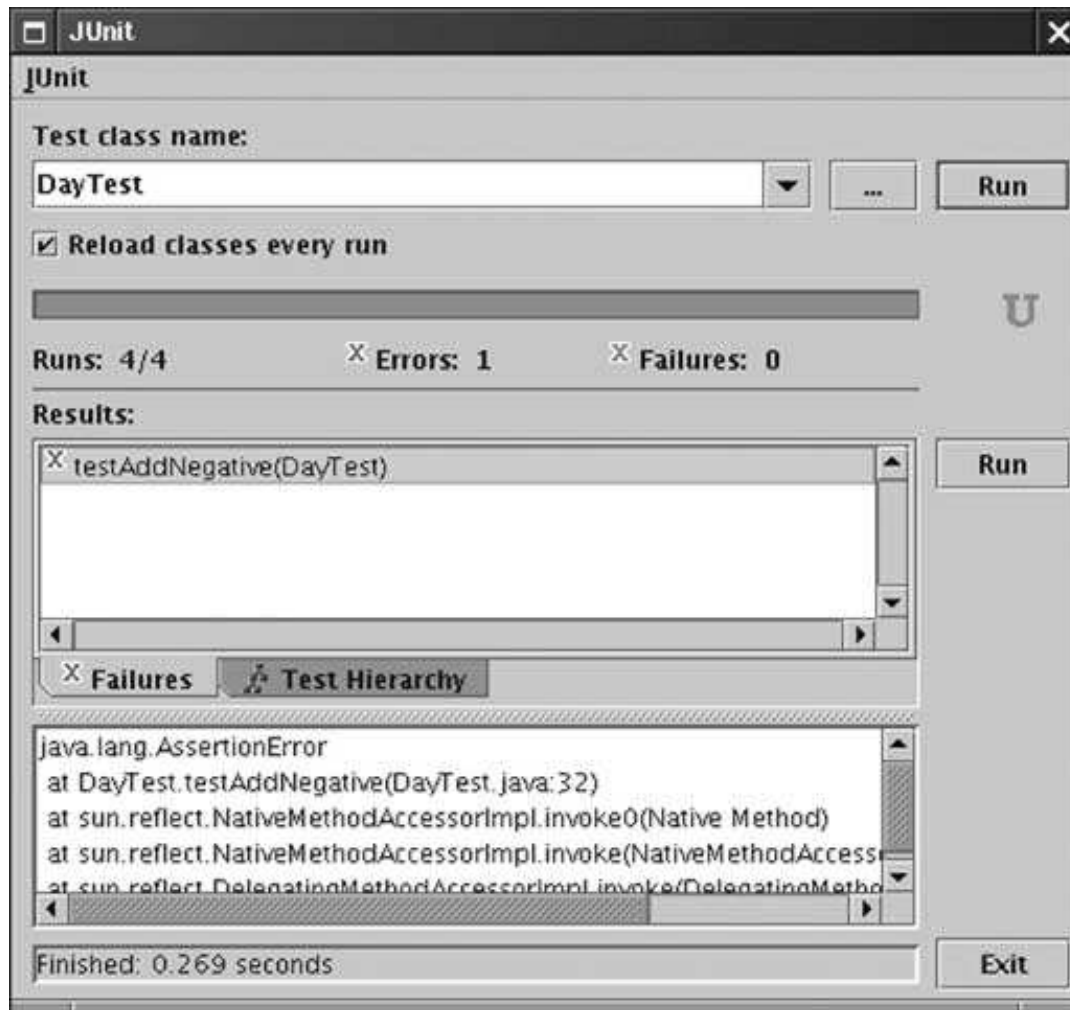
# Unit Testing

- 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

---

[previous](#) | [start](#) | [next](#) .... [Slide 54] ....

# JUnit





---

# JUnit

- Test class name = tested class name + Test
- Test methods start with test

```
import junit.framework.*;
public class DayTest extends TestCase
{
    public void testAdd() { ... }
    public void testDaysBetween() { ... }
    . . .
}
```

---

[previous](#) | [start](#) | [next](#) .... [Slide 56] ....

---

# JUnit

- Each test case ends with assertion
- Test framework catches assertion failures

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

---

[previous](#) | [start](#) .... [Slide 57] ....