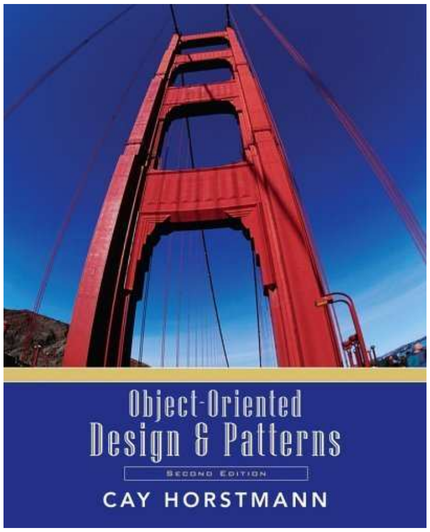
next

Object-Oriented Design & Patterns

Cay S. Horstmann

Chapter 3

The Object-Oriented Design Process



next

previous | start | next

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

previous | start | next

Date Classes in Standard Library

- Many programs manipulate dates such as "Saturday, February 3, 2001"
- `Date` class:

```
Date now = new Date();   // constructs current date/timeSystem.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

previous | start | next

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

previous | start | next

previous | start | next

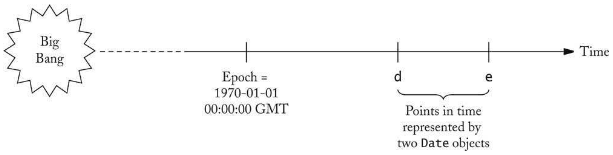
Methods of the `Date` class

- 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

previous | start | next

Points in Time



previous | start | next

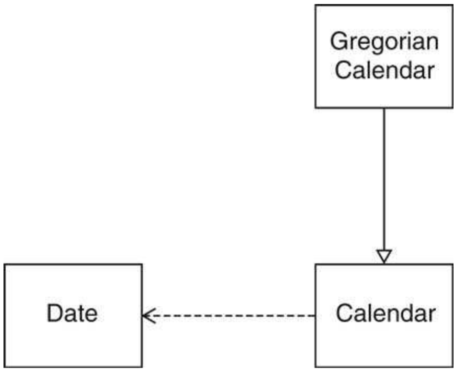
previous | start | next

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

Date Handling in the Java Library



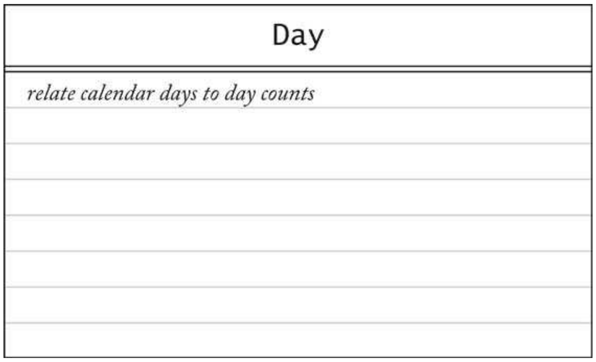
Designing a `Day` Class

- 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

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



Day

relate calendar days to day counts

previous | start | next

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) == nd1.addDays(d2.daysFrom(d1)) == d2
```

- Clearer when written with "overloaded operators":

```
(d + n) - d == nd1 + (d2 - d1) == d2
```

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

previous | start | next

previous | start | next

Implementing a Day Class

- Straightforward implementation:

```
private int yearprivate int monthprivate 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

previous | start | next

previous | start | next

Implementing a Day Class

- Ch3/code/day1/Day.java
- Ch3/code/day1/DayTester.java
- Note private helper methods
- Computations are inefficient: a day at a time

previous | start | next

```
public class Day
{
   /**
      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
      @param aYear a year != 0
      @param aMonth a month between 1 and 12
      @param aDate a date between 1 and 31
   */
   public Day(int aYear, int aMonth, int aDate)
   {
      year = aYear;
      month = aMonth;
      date = aDate;
   }

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

   /**
      Returns the month of this day
      @return the month
   */
   public int getMonth()
   {
      return month;
   }

   /**
      Returns the day of the month of this day
      @return the day of the month
   */
   public int getDate()
   {
      return date;
   }

   /**
      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)
   {
      Day result = this;
      while (n > 0)
      {
         result = result.nextDay();
         n--;
      }
      while (n < 0)
      {
         result = result.previousDay();
         n++;
      }
      return result;
   }

   /**
      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)
   {
      int n = 0;
      Day d = this;
      while (d.compareTo(other) > 0)
      {
         d = d.previousDay();
         n++;
      }
      while (d.compareTo(other) < 0)
      {
         d = d.nextDay();
         n--;
      }
      return n;
   }

   /**
      Compares this day with another day.
      @param other the other day
      @return a positive number if this day comes after the
      other day, a negative number if this day comes before
      the other day, and zero if the days are the same
    */
   private int compareTo(Day other)
   {
      if (year > other.year) return 1;
      if (year < other.year) return -1;
      if (month > other.month) return 1;
      if (month < other.month) return -1;
      return date - other.date;
   }

   /**
      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);
   }

   /**
      Computes the previous day.
      @return the day preceding this day
    */
   private Day previousDay()
   {
      int y = year;
      int m = month;
      int d = date;

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

   /**
      Gets the days in a given month
      @param y the year
      @param m the month
      @return the last day in the given month
    */
   private static int daysPerMonth(int y, int m)
   {
      int days = DAYS_PER_MONTH[m - 1];
      if (m == FEBRUARY && isLeapYear(y))
         days++;
      return days;
   }

   /**
      Tests if a year is a leap year
      @param y the year
      @return true if y is a leap year
    */
   private static boolean isLeapYear(int y)
   {
      if (y % 4 != 0) return false;
      if (y < GREGORIAN_START_YEAR) return true;
      return (y % 100 != 0) || (y % 400 == 0);
   }

   private int year;
   private int month;
   private int date;

   private static final int[] DAYS_PER_MONTH
         = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

   private static final int GREGORIAN_START_YEAR = 1582;
   private static final int GREGORIAN_START_MONTH = 10;
   private static final int GREGORIAN_START_DAY = 15;
   private static final int JULIAN_END_DAY = 4;

   private static final int JANUARY = 1;
   private static final int FEBRUARY = 2;
   private static final int DECEMBER = 12;
}
```

```
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
   }
}
```

previous | start | next

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

```
public class Day
{
   /**
      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
      @param aYear a year != 0
      @param aMonth a month between 1 and 12
      @param aDate a date between 1 and 31
    */
   public Day(int aYear, int aMonth, int aDate)
   {
      julian = toJulian(aYear, aMonth, aDate);
   }

   /**
      Returns the year of this day
      @return the year
    */
   public int getYear()
   {
      return fromJulian(julian)[0];
   }

   /**
      Returns the month of this day
      @return the month
    */
   public int getMonth()
   {
      return fromJulian(julian)[1];
   }

   /**
      Returns the day of the month of this day
      @return the day of the month
    */
   public int getDate()
   {
      return fromJulian(julian)[2];
   }

   /**
      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;
   }

   /**
      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 was 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;
  }

  /**
      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 ca

    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 };
  }

  private int julian;
}
```

previous | start | next

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

```
public class Day
{
  /**
      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
      @param aYear a year != 0
      @param aMonth a month between 1 and 12
      @param aDate a date between 1 and 31
   */
  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;
  }

  /**
      Returns the month of this day
      @return the month
   */
  public int getMonth()
  {
    ensureYmd();
    return month;
  }

  /**
      Returns the day of the month of this day
      @return the day of the month
   */
  public int getDate()
```

```java
  {
    ensureYmd();
    return date;
  }

  /**
      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)
  {
    ensureJulian();
    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)
  {
    ensureJulian();
    other.ensureJulian();
    return julian - other.julian;
  }

  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;
  }

  /**
      Converts this Julian day mumber to a calendar date if necessa
   */
  private void ensureYmd()
  {
    if (ymdValid) return;
    int[] ymd = fromJulian(julian);
    year = ymd[0];
    month = ymd[1];
    date = ymd[2];
    ymdValid = true;
  }

  /**
      Computes the Julian day number of the given day 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;
  }

  /**
      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 ca

    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 };
  }

  private int year;
  private int month;
  private int date;
  private int julian;
  private boolean ymdValid;
  private boolean julianValid;
}
```

previous | start | next

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

previous | start | next

previous | start | next

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

previous | start | next

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); // ERRORdeadline.setDate(28);
      ```

    - Maybe we should call `setDate` first?

      ```
      Day deadline = new Day(2001, 2, 28);deadline.setDate(31); // ERRORdeadline.setMonth(3);
      ```

- `GregorianCalendar` implements confusing *rollover*.
    - Silently gets the wrong result instead of error.
- Immutability is useful

previous | start | next

previous | start | next

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;}
  ```

previous | start | next

Sharing Mutable References

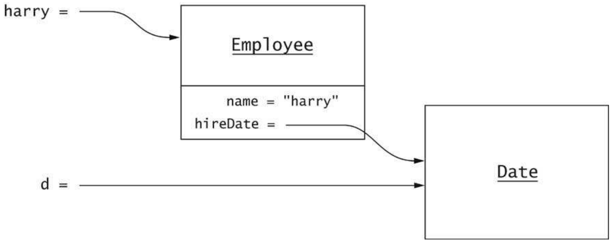
- Pitfall:

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

- Remedy: Use clone

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

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

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:

```
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:

  ```
  String getCurrent();void next();
  ```

- Even more convenient:

  ```
  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
  - 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);
  ```

- Side effect: updates `position` parameter
- 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 beyond implicit parameter

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

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:

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

previous | start | next

Completeness

- Support operations that are well-defined on abstraction
- Potentially bad example: `Date`

```
Date start = new Date();// do some workDate 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

previous | start | next

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

previous | start | next

previous | start | next

Clarity

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

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

- Iterate through list:

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

previous | start | next

previous | start | 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:

```
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?

previous | start | next

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?

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

Programming by Contract

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

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?

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 remove(){   return elements.remove(0);}
```

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

Circular Array Implementation

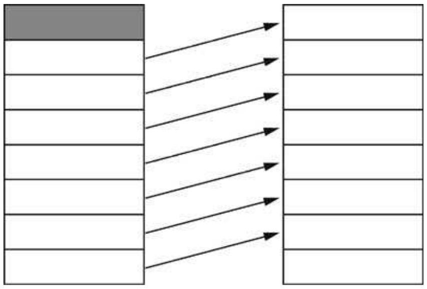
- Efficient implementation of bounded queue
- Avoids inefficient shifting of elements
- Circular: head, tail indexes wrap around
- Ch3/queue/MessageQueue.java

```
/**
    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;
   }

   /**
       Remove message at head.
       @return the message that has been removed from the queue
       @precondition size() > 0
    */
   public Message remove()
   {
      Message r = elements[head];
      head = (head + 1) % elements.length;
      count--;
      return r;
   }

   /**
       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;
}
```

previous | start | next

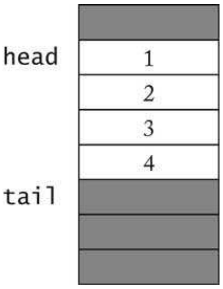
Inefficient Shifting of Elements



previous | start | next

previous | start | next

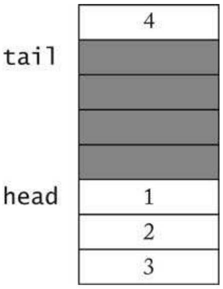
A Circular Array



previous | start | next

previous | start | next

Wrapping around the End



previous | start | next

previous | start | next

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

previous | start | next

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

previous | start | next

Assertions

```
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`

previous | start | next

previous | start | next

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

previous | start | next

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

- First check it’s true for constructor
  - Sets `head = 0`
  - Need precondition `size > 0`!
- Check mutators. Start with `remove`
  - Sets $\mathtt{head}_{\mathtt{new}} \mathtt{ = (head}_{\mathtt{old}} \mathtt{ + 1) \% elements.length}$
  - We know $\mathtt{head}_{\mathtt{old}}$ >= 0 (Why?)
  - `%` operator property:

    $\mathtt{0 <= head}_{\mathtt{new}} \mathtt{ \&\& head}_{\mathtt{new}} \mathtt{ < elements.length}$

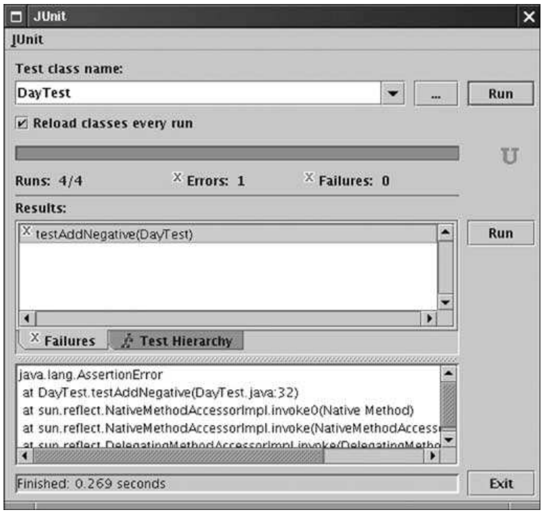
- What’s the use? Array accesses are correct!

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

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



Test class name: DayTest

☑ Reload classes every run

Runs: 4/4 Errors: 1 Failures: 0

Results:

testAddNegative(DayTest)

```
java.lang.AssertionError
 at DayTest.testAddNegative(DayTest.java:32)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccess
```

Finished: 0.269 seconds

JUnit

- Convention: 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() { ... }   . . .}
```

JUnit

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

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