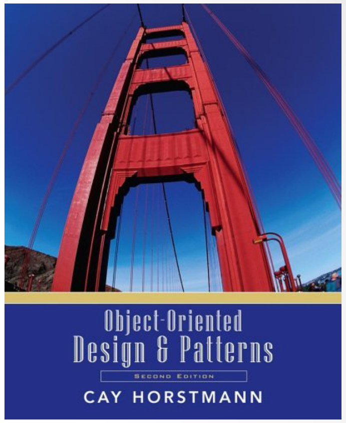
Object-Oriented Design & Patterns

2nd edition

Cay S. Horstmann



Chapter 3: Guidelines for Class Design

CPSC 2100

Software Design and Development

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.

Chapter Objective

- How to find classes for solving a practical programming problem.
- Take very different “bottom up” point of view, and explore how to write a single class well.

Date Classes in Standard Library

- Many programs manipulate dates such as "Saturday, February 3, 2001"
- Date class (java.util):

```
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
```

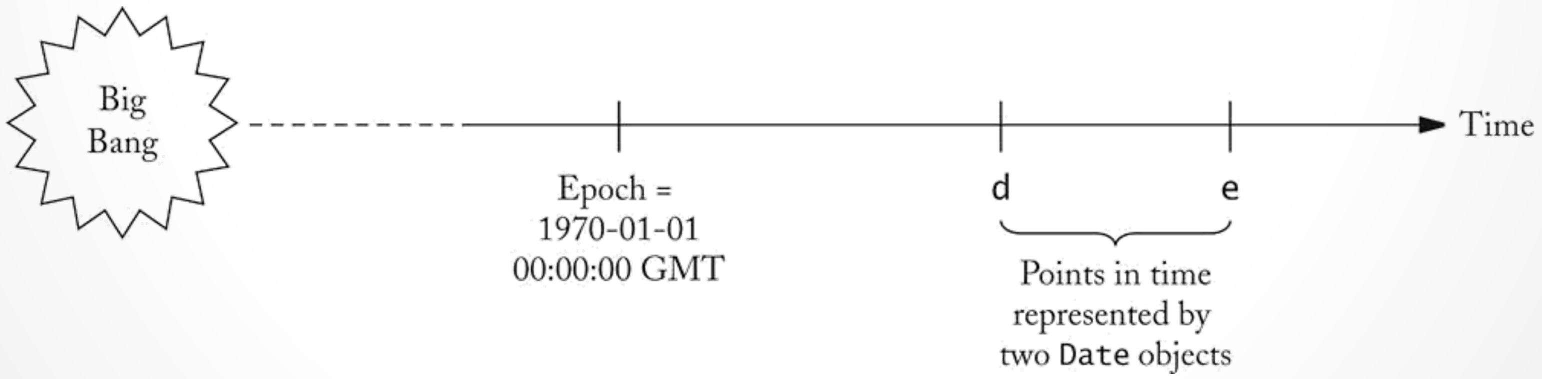
Methods of the Date class

Method	Description
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

- Date class encapsulates *point in time*.
- Date class methods supply *total ordering* on Date objects.
- Convert to scalar time measure.

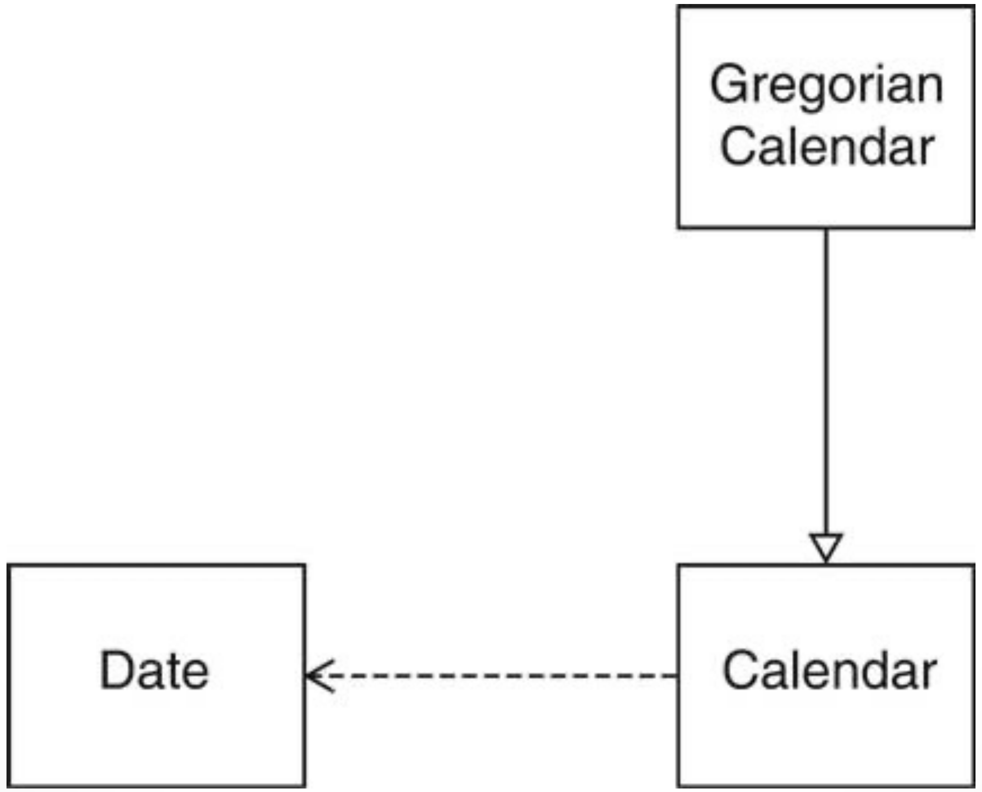
Points in time:


Big Bang
Time
Epoch = 1970-01-01 00:00:00 GMT
d
e
Points in time represented by two Date objects

The GregorianCalender 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



Methods of the Calendar class

Method	Description
int **get**(int field)	Gets a field value; field is a Calendar class constant such as YEAR, MONTH, DATE, HOUR, MINUTE, SECOND
void **set**(int field, int value)	Sets a field value
void **add**(int field, int increment)	Adds to a field value
Date **getTime**( )	Get the Date value
void **setTime**(Date d)	Converts from a Date value

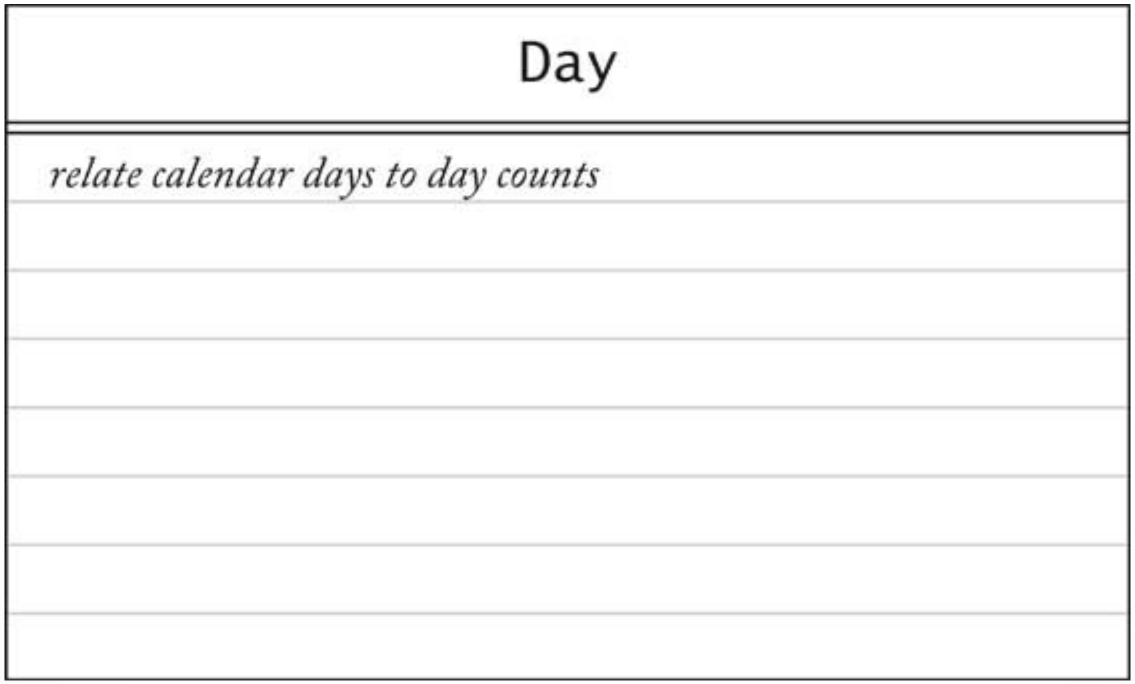
Designing a Day Class

- **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

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);
```

Designing a Day Class

- **Constructor** Date(int year, int month, int date)

- **getYear**, **getMonth**, **getDate** acccesors
- Our Day class has the following public interface.

DayTester.java

```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()
                + "-" + later.getMonth()
                + "-" + later.getDate());
        System.out.println(later.daysFrom(today)); // Prints 999
    }
}
```

Implementing a Day Class

- Straightforward implementation:

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

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

public int getYear( )
{
      return year;
}
.....
```

Implementing a Day Class

- **`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

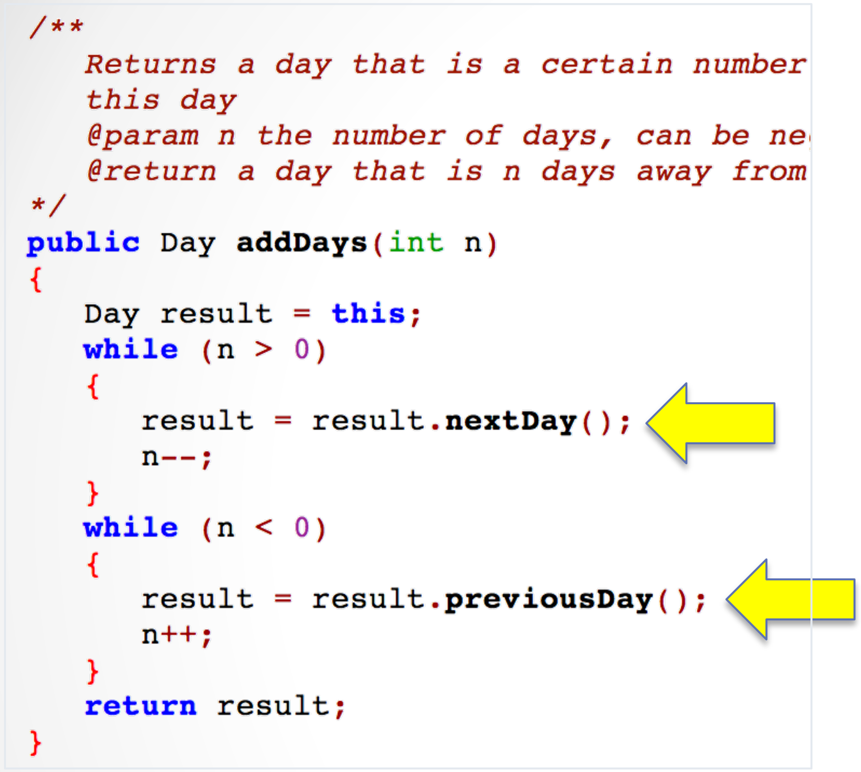
First implementation:

    Day.java

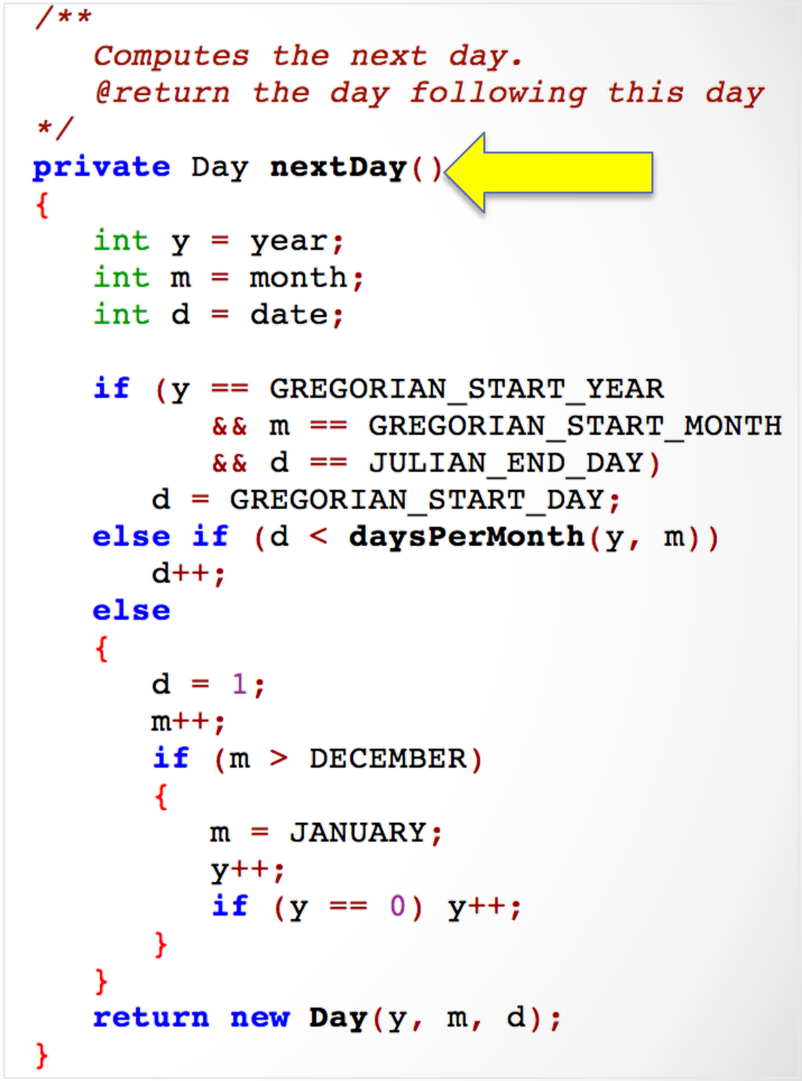
- Note private helper methods.
- Computations are inefficient:

    increment/decrement one day at a time.

Private Helper Methods



```
/**
   Returns a day that is a certain number
   this day
   @param n the number of days, can be ne
   @return a day that is n days away from
*/
public Day addDays(int n)
{
   Day result = this;
   while (n > 0)
   {
      result = result.nextDay();
      n--;
   }
   while (n < 0)
   {
      result = result.previousDay();
      n++;
   }
   return result;
}
```



```
/**
   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);
}
```

Helper methods:

1. Clutter up the public interface.
2. Require special protocol or calling order.
3. Depend on a particular implementation.

Implementation of Date Class

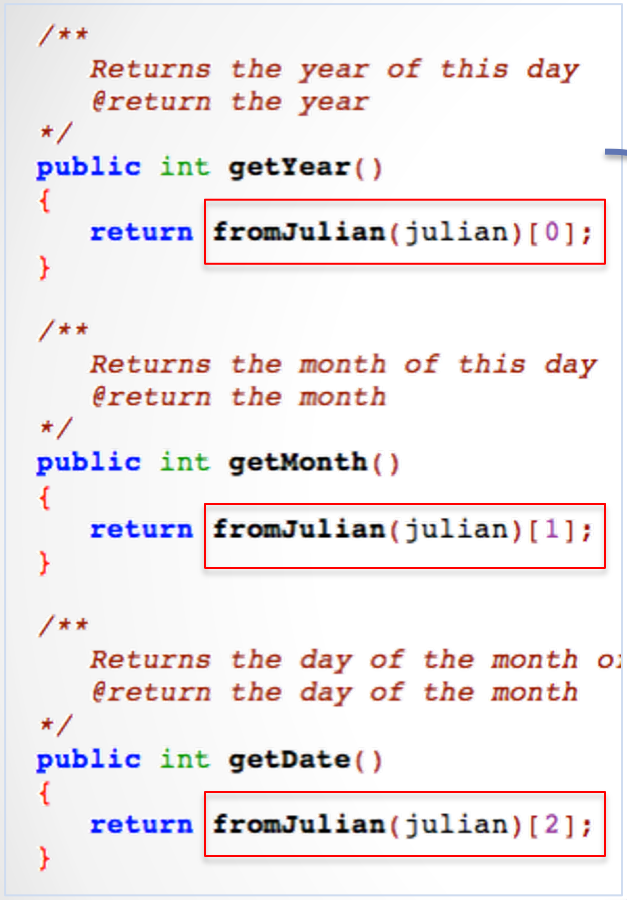
Second implementation:

- For greater efficiency, use Julian day number.
- Used in astronomy.
- Number of days since Jan. 1, 4713 BCE.

Example: May 23, 1968 = Julian Day 2,440,000.

- Greatly simplifies date arithmetic.

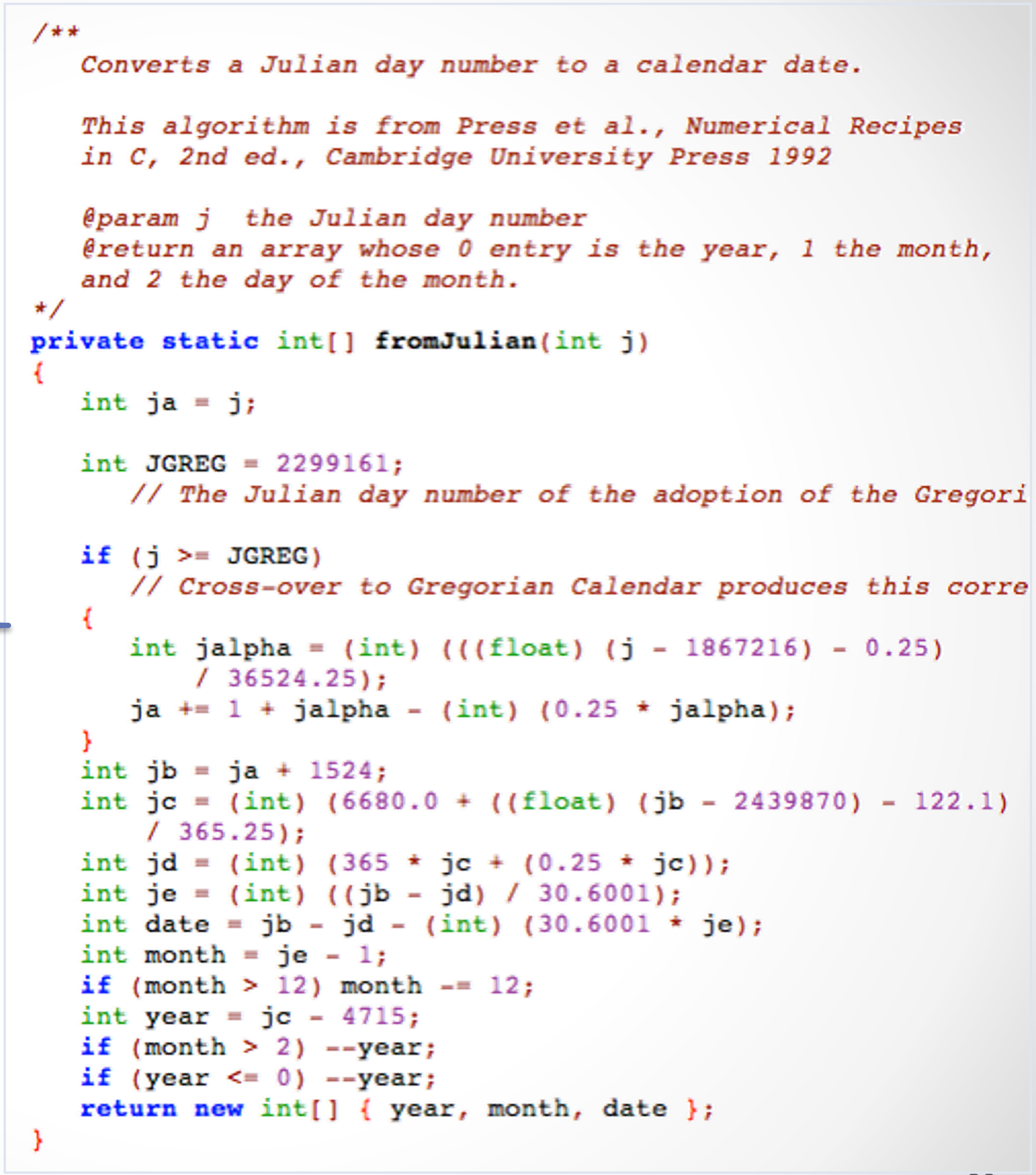
Day.java



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



```
/**
   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 Gregori

   if (j >= JGREG)
      // Cross-over to Gregorian Calendar produces this corre
   {
      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 };
}
```

Implementation of Date Class

Third implementation:

- Second implementation: constructor, accessors are inefficient.
- Best of both worlds: Cache known Julian, y/m/d values.

- Complex and require more storage.

Day.java

- Which implementation is best?

The Importance of Encapsulation

- Even a simple class can benefit from different implementations.
- Users are unaware of implementation.

```
public class Day
{
        . . .
        public int year;
        public int month;
        public int date;
        . . .
}
```

- Public instance variables would have blocked improvement.

Assessors 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

```
    Day deadline = new Day(2001, 1, 31);

    deadline.setMonth(2);

    Day deadline = new Day(2001, 2, 28);

    deadline.setDate(31);
```

- **Immutability is useful.**

Sharing Mutable References

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

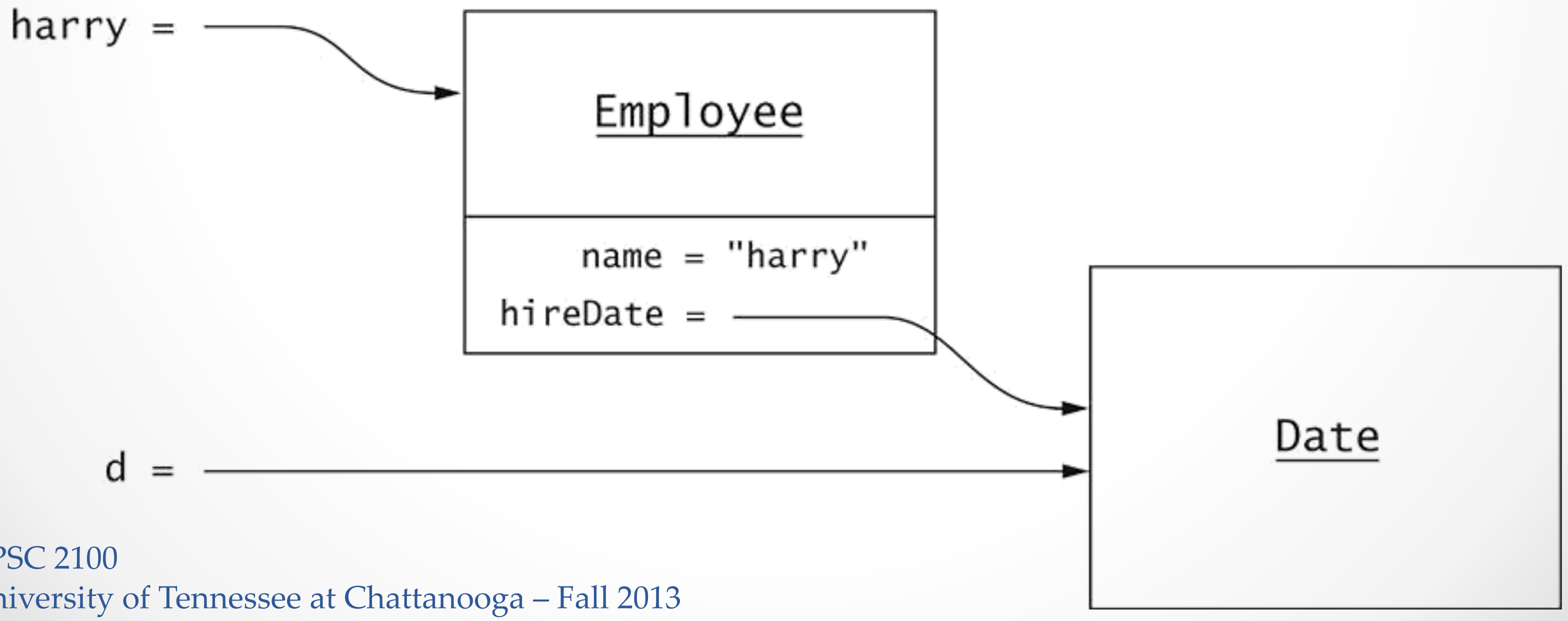
```
class Employee
{
      . . .
      public String getName() { return name; }
      public double getSalary() { return salary; }
      public Date getHireDate() { return hireDate; }

      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!!!
```



Sharing Mutable References

- **Remedy:** Use clone

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

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: BankAccount (getBalance).
- 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();
```

- **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 fields.
- 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.
- **Rule of thumb:** Minimize side effects beyond implicit parameter.

Law of Demeter

- **Example:** Mail system in chapter 2

  ```
  Mailbox currentMailbox = mailSystem.findMailbox(...);
  ```

  return Mailbox object.

  Connection add remove message from Mailbox.

- Breaks encapsulation.

- Suppose future version of MailSystem uses a database.

Law of Demeter

- **The law:** A method should only use:
  - 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);
. . .
```

Quality of Class Interface

- Programmers using the class.
- **Criteria:**
  - Cohesion
  - Completeness
  - Convenience
  - Clarity
  - Consistency

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) { ... }
     ...
}
```

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.

```
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());
```

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(”X”); // AX|BC
```

Clarity

- 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.

Consistency

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

- Bad example:

```
new GregorianCalendar(year, month - 1, day)
```

$$\left.\begin{array}{l}\texttt{- month between 0 and 11}\\ \texttt{- day between 1 and 31}\end{array}\right\}\ \textbf{\textit{not consistent}}$$

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

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.

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**

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

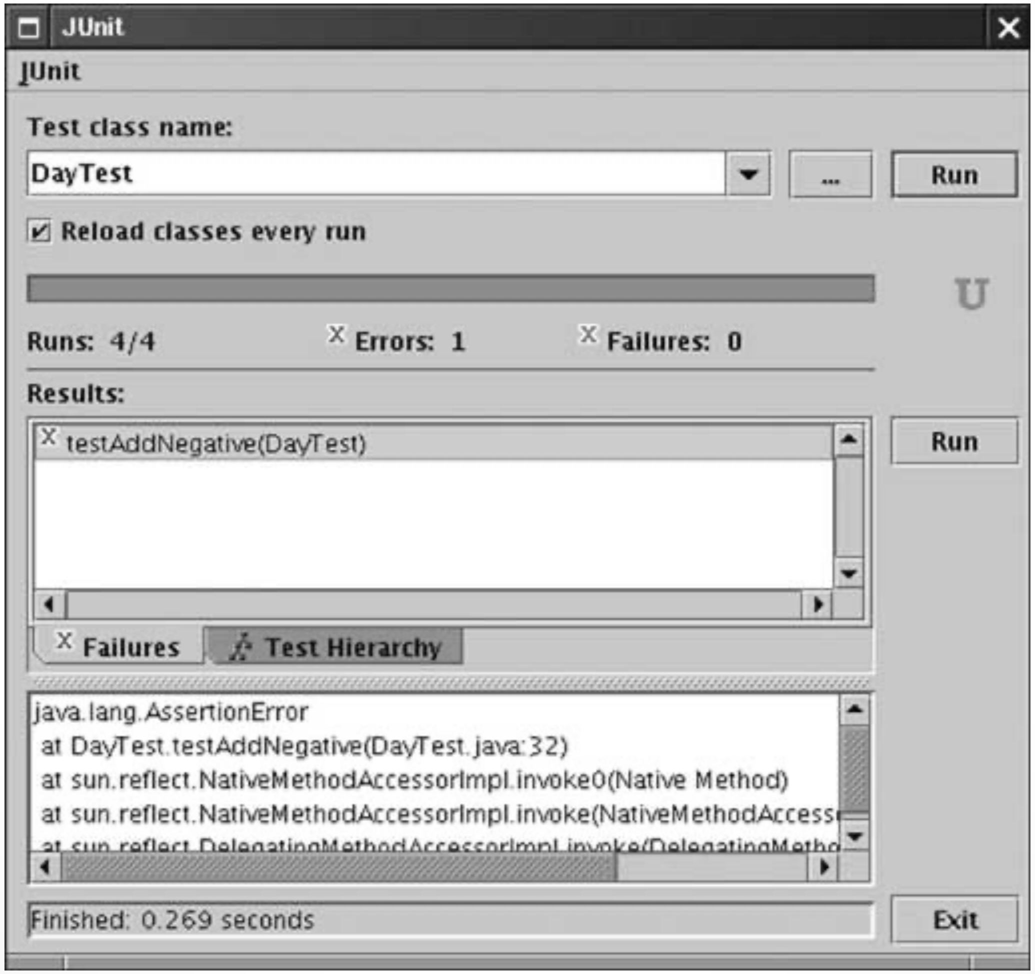
- Condition that is:
  - true after every constructor.
  - preserved by every method.
    (if it's true before the call, it's again true afterwards).

- Useful for checking validity of operations.

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



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

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);
}
```

JUnit 4.x Annotations (1/2)

Annotation	**Description**
@Test public void method( )	The annotation @Test identifies that a method is a test method.
@Before public void method( )	Will execute the method before each test. This method can prepare the test environment (e.g. read input data, initialize the class).
@After public void method( )	Will execute the method after each test. This method can cleanup the test environment (e.g. delete temporary data, restore defaults).
@BeforeClass public void method( )	Will execute the method once, before the start of all tests. This can be used to perform time intensive activities, for example to connect to a database.

http://www.vogella.com/articles/JUnit/article.html

JUnit 4.x Annotations (2/2)

Annotation	Description
@AfterClass public void method( )	Will execute the method once, after all tests have finished. This can be used to perform clean-up activities, for example to disconnect from a database.
@Ignore	Will ignore the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included.
@Test (expected = Exception.class)	Fails, if the method does not throw the named exception.
@Test(timeout=100)	Fails, if the method takes longer than 100 milliseconds.

Assert Statements (1/2)

Statement	**Description**
fail(String)	Let the method fail. Might be used to check that a certain part of the code is not reached. Or to have failing test before the test code is implemented.
assertTrue(true) / assertTrue(false)	Will always be true / false. Can be used to predefine a test result, if the test is not yet implemented.
assertTrue([message], boolean condition)	Checks that the boolean condition is true.
assertsEquals([String message], expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertsEquals([String message], expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.

Assert Statements (2/2)

Statement	**Description**
assertNull([message], object)	Checks that the object is null.
assertNotNull([message], object)	Checks that the object is not null.
assertSame([String], expected, actual)	Checks that both variables refer to the same object.
assertNotSame([String], expected, actual)	Checks that both variables refer to different objects.
assertNull([message], object)	Checks that the object is null.

End of Chapter 3