Lecture9 面向对象设计&模式

1. Java 库中的 Date 类概述

Date 类

Date 类封装了时间点

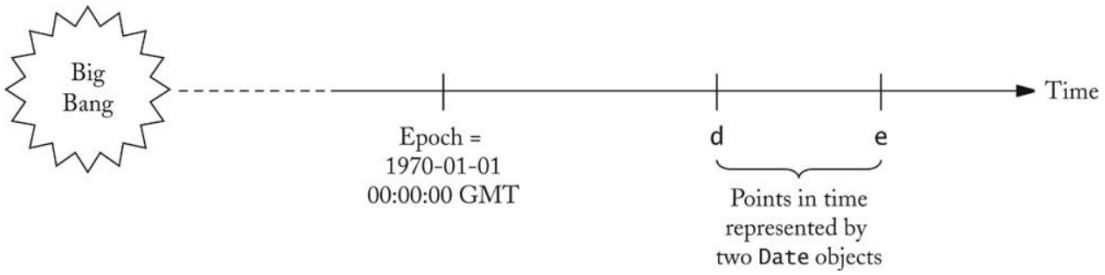
```
// constructs current date/time
Date now = new Date();
System.out.println(now.toString());
// prints date such as
// Sat Feb 03 16:34:10 PST 2001
```

Date 类的方法

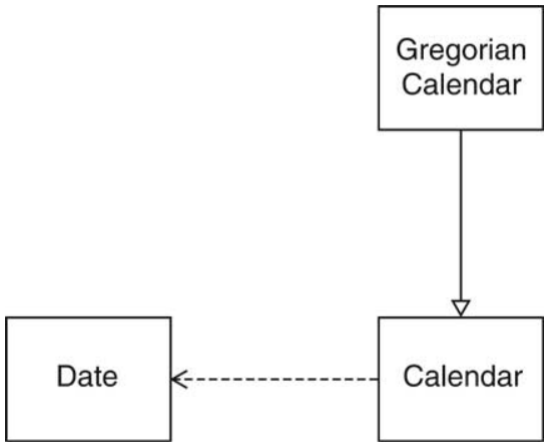
`boolean after(Date other)`	Tests if this date is after the specified date
`boolean before(Date other)`	Tests if this date is before the specified date
`int compareTo(Date other)`	Tells which date came before the other
`long getTime()`	Returns milliseconds since the epoch (1970-01-01 00:00:00 GMT)
`void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

- 具有一些弃用方法 Deprecated methods
- Date 类方法提供任何两个 Date 对象的排序功能（比较时间大小）
- 转换为标量时间度量

Date 的时间点 Points in time



GregorianCalendar 类



Date 类不度量月份、工作日等，用 Calendar 类处理

日历将一个名称分配给一个 point in time

有很多在使用的 Calendar 类

- Gregorian
- 当代：Hebrew, Arabic, Chinese
- 历史上：French Revolutionary，Mayan

2. 设计一个 Day 类

要求

- 一个定制的类
- 在自己的程序中使用标准库类，而不是这个类
- Day 类将一天封装在一个固定的位置
- 没有时间，没有时区
- 使用公历

目标方法

`dayFrom()` 方法计算两天之间的天数

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

`addDays()` 方法计算离给定日期数天的某一天

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

一些数学关系

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

如果使用重载运算符更加清楚

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

构造函数

```
Day(int year, int month, int date)
```

访问方法

```
getYear, getMonth, getDate
```

3. 三种 Day 类的实现

- 版本1：Day.java.html
  - 容易实现
  - 计算效率很低
- 版本2：Day.java.html
  - 使用 Julian Day 大大简化计算量
  - 构造函数，访问器效率很低
- 版本3：Day.java.html
  - 使用一些 Cache 的技巧减少不必要的计算

4. 封装的重要性

- 即使是一个简单的类也可以从不同的实现中受益
- 用户不知道实现
- 公共实例变量会阻碍改进
- 不要使用公共字段，即使对于“简单”的类也是如此

```
d.year;
d.getYear();

d.year++;
d = new Day(d,getDay(), d.getMonth(), d.getYear()+1);
```

访问器 Accessors 和赋值器 Mutators

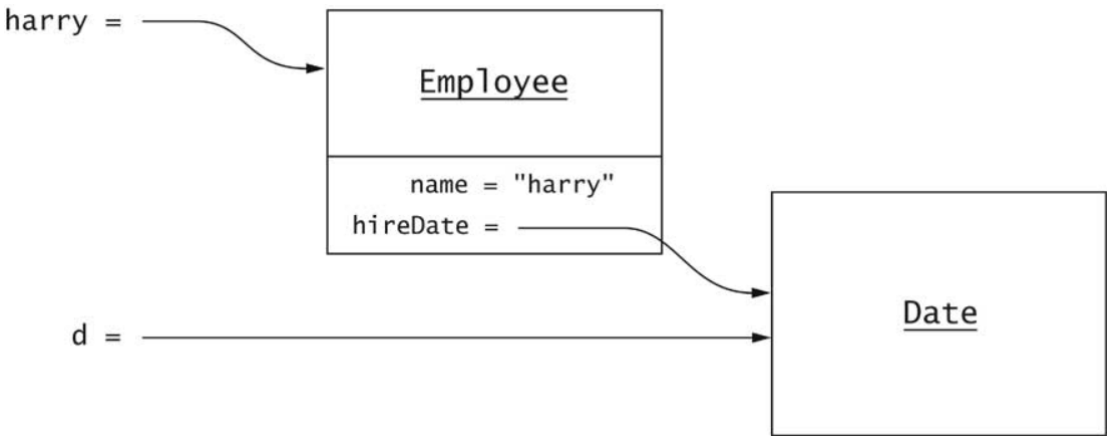
- Mutator：改变对象的状态
- Accessor：读取对象的状态，但是不改变它
- Day 类是没有赋值器的
- 一个没有赋值器的类被称为不变的 immutable
- String 类是不变的
- Date 和 GregorianCalendar 是可变的

共享可变的引用

- 对**不可变对象**的引用可以**自由共享**
- 不要共享可变引用

```
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;
}
```

陷阱




```
Employee harry = . . .;
Date d = harry.getHireDate();
d.setTime(t); // 对d的改变事实上也改变了Harry的状态!!!
```

补救：使用克隆

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

Final 实例字段

- 将**不可变实例字段**标记为 `final` 是个好主意
    - ```
      private final int day;
      ```
- final 对象的引用**仍然可以指向可变的对象**
    - ```
      private final ArrayList elements;
      ```
    - 元素不能引用另一个数组列表
    - 数组列表的内容可以更改

分离访问器和赋值器

如果我们调用一个方法来访问一个对象，我们不希望对象发生变化

经验

- 赋值器通常返回 `void`

一种很好的接口定义

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

下面的定义更加的方便

```
String getCurrent();
String next(); // returns current
```

赋值器可以返回一个方便的值，前提是还有一个访问器可以获得相同的值

副作用

- 一个方法如果可以改变可观察 / 可调用的状态，那么它可能有副作用
- 赋值器：改变隐式参数
- 其它的副作用：
    - 显式参数的更改
    - static 对象的更改
- 尽可能避免这些副作用，它们影响使用者

一个好的例子：隐式的参数不会受到影响

```
a.addAll( b );
```

- a 会改变，但是不会影响 b

一个坏的例子

```
SimpleDateFormat formatter = . . .;
String dateString = "January 11, 2012";
FieldPosition position = . . .;
Date d = formatter.parse( dateString, position );
```

- 副作用：更改了 `position` 参数

最少知识原则 Law of Demeter

一个方法应该只使用下面的对象

- 用 new 构造的类
- 参数
- 类里面的实例字段

不应该使用从方法调用返回的对象

5. 分析接口的质量

评判标准

- 客户：使用该类的程序员
- 评判标准
  - 内聚性
  - 完整性
  - 便捷性
  - 明确性
  - 一致性
- 工程活动：权衡

内聚性 Cohesion

- 类描述一个单独的抽象
- 方法应该与单个抽象类相关联

一个很差的示例

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

完整性 Completeness

- 支持在抽象上定义良好的操作

一个很差的示例

- Java Date 类不具备计算经过了多少毫秒的这种方法
- 这不在责任范围之内吗
- 毕竟，我们有 `before`，`after` 和 `getTime` 方法

便捷性 Convenience

一个好的结构可以让所有的任务都成为可能

一个很差的示例

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

- 为什么 `System.in` 没有一个 `readLine()` 方法?
- 毕竟 `System.out` 有 `println()` 方法
- Scanner 类修复了不便

明确性 Clarity

困惑的程序员会编写出错误代码

一个很差的示例：从 LinkedList 中删除元素

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

迭代 LinkedList

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

- 元素之间的迭代器，就像文字处理器里闪烁的插入符号
- `add()` 添加到迭代器的左边（像字处理器）

把 X 添加到 B 前面

```
ListIterator<String> iterator = countries.listIterator();  // |ABC
iterator.next();    // A|BC
iterator.add( "France" );    // AX|BC
```

然而 `remove()` 不会删除迭代器左边的元素

- API：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.

一致性 Consistency

类的相关特性应该具有匹配性

- 名称
- 参数
- 返回值
- 行为

一个很差的示例

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

- 为什么月份是从 0 开始记的?

6. 契约式编程 Programming by contract

- 说明职责
  - 对于调用者的
  - 对于实现者的
- 增加可靠性
- 增加效率

先决条件

- 调用者试图从空 MessageQueue 中删除消息
- 会发生什么?
  - MessageQueue 可以将此声明为错误：过度的错误检查代价高昂
  - MessageQueue 可以容忍调用并返回哑值：返回哑值会使测试复杂化
- 以合同为比喻
  - 服务提供者必须指定前置条件
  - 前提条件满足时，服务提供者必须正常工作
  - 否则，服务提供者可以做任何事情
- 当前置条件失败时，服务提供者可能会
  - 抛出异常
  - 返回错误答案
  - 损坏数据

断言 Assertions

- 一种警告程序员的机制
- 可以在测试之后关闭
- 有用的警告程序员的先决条件失败
- 如果 condition 为假并且启用了检查，则抛出 `AssertionError`

语法

```
assert condition;
assert condition : explanation;
```

后置条件

- 服务提供者保证的条件
- 每个方法都承诺描述，@return
- 有时，可以断言附加的有用条件

类的不变式 Class Invariants

若类中的一个条件满足

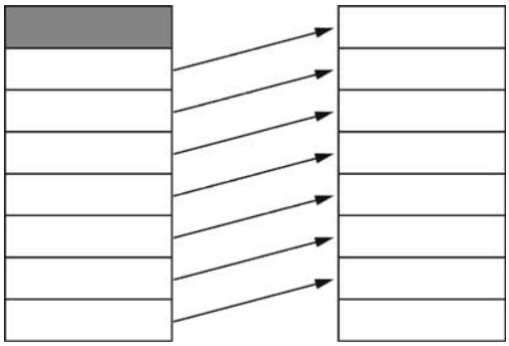
- 在每次构造后始终为真
- 在每一次方法的调用中都保持不变（例如始终为真）

则它称为类的不变式，对于判断操作的有效性很有用

示例 —— 环形数组

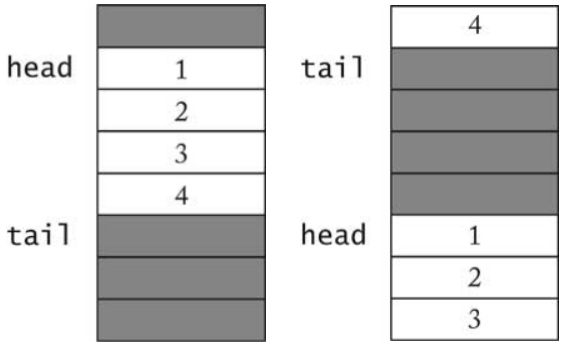
环形数组介绍

常规数组中，移除一个元素需要 $O(n)$ 的时间复杂度



- 效率很低

环形数组示意如下



- `head`：指向第一个不为空的元素
- `tail`：指向第一个为空的元素

先决条件

- 环形数组的 `add` 需要先决条件

```
@precondition size() < elements.length
```

```
@precondition size() < getCapacity() // 更好的选择
```

断言

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

契约抛出异常

```
/**
   . . .
   @throws NoSuchElementException if queue is empty
*/
public Message remove () {
   if (count == 0)
      throw new NoSuchElementException();
   Message r = elements[head];
   . . .
}
```

- 异常抛出契约的一部分
- 调用者可以依赖于行为
- 异常抛出不是违反先决条件的结果
- 该方法没有前置条件

后置条件

在 `add` 方法添加后置条件

```
@postcondition size() > 0
```

一个调用的后置条件可以暗示另一个调用的前置条件

```
q.add(m1);
m2 = q.remove();
```

类的不变式

在环形数组的构造中，存在

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

首先，在构造器中，有

- `Sets head = 0`
- 需要前置条件 `size > 0`

在调用方法 `remove` 的时候

- $head_{new}$ = ($head_{old}$ + 1) % elements.length
- 我们知道 $head_{old}$ > 0
- 可得到 0 <= $head_{new}$ && $head_{new}$ < elements.length

7. 单元测试

- 单元测试 = 单个类的测试
- 在实现过程中设计测试用例
- 在每次实现更改之后运行测试
- 当发现一个 bug 时，添加一个捕获它的测试用例

名称规范

- 测试类的名称 = 被测试的类名称 + Test
- 测试的方法以 `test` 开头

```
1  import junit.framework.*;
2  public class DayTest extends TestCase {
3     public void testAdd () { ... }
4     public void testDaysBetween () { ... }
5     . . .
6  }
```

- 每个测试用例都以 `assertTrue` 方法结束，或者是其它的单元测试方法如 `assertEquals`
- 测试框架捕获断言失败

单元测试示例

```
1  public void testAdd () {
2     Day d1 = new Day( 1970, 1, 1 );
3     int n = 1000;
4     Day d2 = d1.addDays( n );
5     assertTrue( d2.daysFrom( d1 ) == n );
6  }
```