CS193J: Programming in JAVA

January 15, 1999

Lecture #4

Announcements

- Webclass registration closed yesterday, TA assignments mailed last night
- First assignment out today
  - Due a week and a half later, but don't postpone getting started!
    - Several classes to write
    - Requires investigation into the built-in classes
    - Need to futz a bit with HTML
- Reading
  - Ch 8 on inheritance
- No lecture on Monday, MLK holiday

Early thoughts on object design

- Never let object get into malformed state
  - No public instance variables
  - Always set up state properly in ctor and/or defaults values
  - Only provide setters if needed, be sure properly constrained
  - Don't hand out references to mutable internal data
- Object should be responsible for own behavior
  - Interface should include complete set of needed methods
  - Need to print a Time? Add print method to class, don't pull out the hour/minute fields and do it yourself
  - Same for converting a string to Time, comparing two times, shifting a time forward, etc.
  - Does every ivar need a setter? A getter?

Internal vs external representation

- To client, might be handy that Time works in terms of hours and minutes
- But this is a difficult representation to manipulate

  What about AM/PM?

  How to add delta to time, how to handle wrap?
- Even a simple operation like before can be quirky

```
boolean before(Time other) {      //note access to other
  return ((this.hour < other.hour) ||
          (this.hour == other.hour &&
                this.minute < other.minute);
}
```

- Is there a better way?

More on representation

- If Time stored in military 24-hour time?
  - Somewhat easier to shift, avoids problems with AM/PM
- If internally tracked as minutes since midnight..?
  - Trivial to implementation "before" operation
  - Trivial to shift, easy to handle wrap around as well
- Can still provide accessor for hour/minute if needed
  - Simple to compute from internal representation
- Does changing the internal data affect client of Time class?
  - What impact does making things public have on implementation flexibility?
  - What is "deprecated" about anyway?

Java libraries

- OOP is supposed to excel in code reuse
  - The class is good unit for packaging functionality
- Java makes good on this promise by providing a large collection of useful built-in classes
  - Your job becomes learning what's there and how to use it, not having to build it in the first place!
- We'll skim over a few of the common classes
  - You'll pick up more from reading text/online docs and experimenting
- All class specifications viewable with web browser
  - See link to Sun's site in "other materials" area of our web site

Packages

- Collections of classes are grouped into "packages"
  - Logical arrangement of related functionality
  - Equivalent of a C library
- Standard class packages: java.lang, java.util, java.net, java.awt, etc.
- Packages help sort out name clashes
  - Every class has unique "full name": java.lang.String, java.awt.Window, java.net.URL
- Package prefix gives path to find class files
  - Each '.' in package name translates to a level of sub-directory
  - CLASSPATH variable starts the search path
  - VM loads classes one at a time as they are referenced

Import

- Somewhat misunderstood
- The **only** thing import does is allow you to use shorthand name, without full package prefix

```
import java.util.Vector;
Vector v = new Vector();
```

Instead of:

```
java.util.Vector u = new java.util.Vector();
```

- Can import all classes in a package with wildcard

```
import java.util.*;
```

Makes everything in the java.util package accessible by shorthand name: Date, Hashtable, etc.

The java.lang package

- Contains classes such as:
  - Object
  - String, StringBuffer
  - Boolean, Byte, Short, Integer, Long, Float, Double
  - Math
  - Exception
  - Thread
  - System, Runtime
- Everything in java.lang already available by short name, no import necessary

Object

- Object is the root class
  - Everything eventually is related to Object
  - Defines the basic functionality all objects have
- Has small set of methods

```
boolean equals(Object other);
long hashCode();
String toString();
```

  plus a few others
- Method toString converts object to string representation, usually for printing

```
System.out.println(someObject);
```

  Calls **`someObject.toString()`** and then prints string result

Primitive wrapper classes

- Byte, Boolean, Short, Integer, Long, Float, Double
  - Rely on Java's case-sensitivity to be different from primitive names
  - Each is object "wrapper" class for a primitive type
  - Designed to be immutable, value set in ctor, no setters
- Sample usage

```
Integer i = new Integer(15);
Float f = new Float("3.15169");
int k = i.intValue();
double d = f.floatValue();
```

- Needed in cases where an object is required
  - Where a primitive type just won't do

String

- Strings are immutable
  - Characters set in constructor of String, never changed
  - Methods that return String objects create new ones to return, do not change receiver String
  - Avoids problems with aliasing, but can be source of inefficiency
- Representative methods:

```
int length()
char charAt(int index)
int indexOf(char ch)
int indexOf(String s)
boolean equals(String s)
String toLowerCase()
String replace(char oldChar, char newChar)
String concat(String)
String substring(int offset, int endIndex)
```

- Convenience features: double-quotes, +, +=

StringBuffer

- Mutable sequence of characters, can grow/shrink
- More efficient for repeated string manipulation
- Representative methods:

```
void setLength(int len)
void setCharAt(int index, char newChar)
StringBuffer append(String s)
StringBuffer insert(String s, int index)
StringBuffer reverse(String s)
```

- Used internally by compiler

```
"You won $" + 100 + "!"
```

Compiles to

```
new StringBuffer().append("You won $").
                   append(100).append("!").toString();
```

The java.util package

- Useful utility classes
- Contains such classes as:
  - Calendar, Date
  - Vector, Stack
  - Dictionary, Hashtable
  - Random
  - StringTokenizer
- This package not visible by default, need to use full name or add import statement

Vector class

- For collections that grow and shrink, rearrange, etc.
  - Implemented internally as an array
  - Throws it out and copies to new one when needed
- How does it work for any type of element?
  - Designed to store things of **Object** type
  - What about primitive types like int and boolean?
- Representative methods

```
int size()
Object elementAt(int index)
int indexOf(Object o)
void addElement(Object o)
void insertElementAt(Object o, int index)
void removeElement(Object o)
```

Working with a Vector

- Adding elements to a Vector

```
Vector colors;

colors = new Vector();
colors.addElement("Yellow");
colors.addElement("Green");
```

- Getting things out of a Vector

```
for (int i = 0; i < colors.size(); i++) {
    String color = (String)colors.elementAt(i);
    System.out.print("Color is " + color);
}
```

- Why do we need the typecast?

  If objects always remember what they are, why need typecast?

  Is the typecast safe?

Searching a Vector

- How does the Vector contains method work?

```
Vector names;
// create and fill Vector with name strings
If (names.contains("Gerhard"))
   ...
```

- There are no function pointers in Java (at this time)

  How do we know if two objects are equal?

  What about two objects being "descriptively equal"?

OOP: let the object decide!

- The object decides for itself what it is equal to

```
public boolean contains(Object key) {
    for (int i = 0; i < size(); i++) {
        if (key.equals(elementAt(i))
            return true;
    }
    return false;
}
```

- The equals method can be implemented in each class to do whatever is the appropriate thing

  Polymorphism in action!

Hashtable object

- Stores key/value pairs for fast lookup
- How is it implemented?
  - Hashes key object
  - Resolves collisions by chaining
  - Buckets are stored as linked lists
  - Increases number of buckets and rehashes when load is too high
- Representative methods

```
int size()
void put(Object key, Object value)
Object get(Object key)
void remove(Object key)
```

- Uses Object as type for key and value
  - So may need typecast on result returned from get

Working with a Hashtable

- Entering into a Hashtable

```
Hashtable directory;

directory = new Hashtable();
directory.put(”Julie”, “zelenski@cs.stanford.edu”);
directory.put(”Matt”, “mvaska@heartport.com”);
```

- Looking up entries

```
String address;
address = (String)directory.get(“Matt”);
System.out.println(“E-mail is “ + address);
```

  Note the typecast to show which kind of object we are getting