CS 61B Data Structures and Programming Methodology

July 3, 2008

David Sun
Announcements

• Project 1 is out! Due July 15\textsuperscript{th}. Check the course website.

• Reminder: the class newsgroup ucb.class.cs61b should be considered as required reading.
Today

- Object
- equals()
- Packages
Object

- **Object class is the ultimate ancestor** – every class in Java extends from the `Object` class.

```
    class Human extends Object

    class Employee extends Human

    class Student extends Human
```

```
    public class Employee
    {
        // Employee class definition
    }

    public class Student
    {
        // Student class definition
    }
```
Object

- A variable of class Object can be used as a generic placeholder for a reference to an object of any type.
  
  ```java
  Object obj = new Employee("Hacker Joe", 35000);
  ```

- To do anything specific with `obj`, you need to cast it to some type:
  
  ```java
  ((Employee)obj).raiseSalary(0.4);
  ```

- All array types extend the object class:
  
  ```java
  Employee[] staff = new Employee[10];
  obj = staff;
  obj = new int[10];
  ```
From Lecture 3:

• **Key characteristics:**
  
  – Object’s Behavior: what can you do with the object, what methods can you call?
  
  – Object’s State: how does the object react to methods applied to it? What does it look like?
  
  – *Object’s Identity: how to distinguish the object from others with same behavior and state?*
equals()

- **A method in the **Object** class that tests whether one object is considered equal to another.**

Reference based equality (default implementation in Java)

```
x
y
```

State-based equality

```
Employee
name = "dave"
id = 101
```

```
Employee
name = "dave"
id = 101
```
equals()

class Employee {
    
    public boolean equals(Object otherObject) {
        //a quick test to see if two objects are equal
        if (this == otherObject) return true;

        //must return false if the explicit parameter is null
        if (otherObject == null) return false;

        //if the classes don’t match they can’t be equal
        if (getClass() != otherObject.getClass())
            return false;

        //now we know otherObject is a non-null employee
        Employee other = (Employee) otherObject;

        //test whether the fields have identical values
        return name.equals(other.name) &&
                salary == other.salary &&
                hireDate.equals(hireDate);
    }
}
equals() in Subclass

- When defining the equals method of a subclass, first call the superclass equals, then compare the instance fields of the subclass:

```java
class Manager{
    ...  
    public boolean equals(Object otherObject) {
        if (!super.equals(otherObject)) return false;   
        //super.equals checked that this and otherObject are in the same  
        //subclass  
        Manager other = (Manager) otherObject;  
        return bonus = other.bonus;  
    }
}
```
A Common Mistake

```java
public class Employee {
    public boolean equals(Employee other) {
        return name.equals(other.name) &&
               salary == other.salary &&
               ID == other.ID;
    }
}

• The explicit parameter other is declared to be Employee, not Object. So
public boolean equals(Employee other) will not override the method
public boolean equals(Object other)

• The compiler and the run time will not pick this up because equals(Employee other) is treated using method overloading (not overriding).
```
@Override

- Since Java 5.0, you can protect yourself against this type of error with the metadata tag `@Override`:

  ```java
  @Override public boolean equals(Object other)
  ```

- If you make a mistake and the new method does not override any method in the superclass then the compiler will pick up the error.

  ```java
  @Override public boolean equals(Employee other)
  ```
toString()

- Another method in Object that returns a string representation of the value of this object.

- When you do:
  
  ```java
  Employee x = new Employee(...);
  System.out.println(x); //calls the toString() method on x returns a //string object
  to //System.out.println
  ```

- Default implementation in Java prints the class name and the hash code of the object:
  
  ```java
  System.out.println(System.out)
  java.io.PrintStream@2f6684
  ```
Packages

• Java organizes “related” of classes into collections called *packages*.

• Packages are used to separate your work from code provided by others:
  
  – Guarantee the uniqueness of class names.
  
  – Packages can contain hidden classes that are used by the package but are not visible or accessible outside the package.
  
  – Classes in packages can have fields and methods that are visible by all classes inside the package, but not outside.
Examples of Packages

• The Java Standard Library is contained in java and javax;
• In the standard library:
  – java.lang;
  – java.util,
  – java.net
  – A nesting/hierarchy of classes like the nested subdirectories on your harddrive.
• Package names are hierarchical.
  java.awt.image.Model refers to the class Model inside the package image inside the package awt inside the package java.
Using Packages

• A class can use all classes from its own package and all public classes from other packages.

• Two ways to access public classes from other packages:
  1. Using a fully qualified name:
     ```java
     java.lang.System.out.println("Here is an example");
     ```
  2. Using the import statement to give you a shorthand way of referring to files in the packages.
     ```java
     import java.util.ArrayList;
     import java.util.*; //now you can refer to everything in .util
     ```

• Every Java program implicitly imports `java.lang.*`, so you don't have to import it explicitly to use `System.out.println()`.
Naming Conflict

• If you import multiple packages that contain classes with the same name, you’ll get an error:

```java
import java.util.*;
import java.sql.*
Date today;  //ERROR
```

• Two solutions:

1. ```java
   import java.util.*;
   import java.sql.*
   import java.util.date.*;
```  
2. ```java
   java.util.Date birthday= new java.util.Date();
```
Static Import

- Java 5.0 introduced syntactic sugar to abbreviate calls to Static (class) methods such as `System.out` and `Math.pow`:
  - `import static java.lang.System.out;` means “within this file, you can use `out` as an abbreviation for `System.out`.
  - `import static java.lang.System.*;` means “within this file, you can use any static method in `System` without mentioning the package."
Adding a Class to a Package

```java
package cs61b.class.day8

double class Employee {  
    ...
}

• If you don’t specify the package statement
  – The classes in that source belongs to the default package.
  – The default package has no package name.

• If you specify the package statement
  – You must place files in a package into a subdirectory that
    matches the full package name.
  – Employee.class must appear in the directory cs61b/class/
    day8 (or cs61b\class\day8 on Windows).
```
How does **javac** and **java** Locate Files?

- They can appear in any of the directories on your CLASSPATH environment variable (in Unix).
  
  - `% printenv CLASSPATH`  
    - `./home/ff/cs61b/lib/ucb.jar`
  
  - Java first looks in ".", the current directory, and then looks in /home/ff/cs61b/lib/ucb.jar (which is a file of classes in a compressed format.
  
  - The CLASSPATH does not include the location of the Java standard library packages (those beginning with java). The Java compiler knows where to find them.
javadoc

• When compiling a source file, the compiler often needs information about a type (a class or interface) it does not yet recognize.
• Javac will search through the directories in the CLASSPATH:
  – *Search produces a class file but no source file:* javac uses the class file.
  – *Search produces a source file but no class file:* javac compiles the source file and uses the resulting class file.
  – *Search produces both a source file and a class file:* javac determines whether the class file is out of date. If the class file is out of date, javac recompiles the source file and uses the updated class file. Otherwise, javac just uses the class file.
package MyList;
public class ListNode {
    int item;
    ListNode next;
}

package MyList;
public class List {
    . . .

    public void insertTail(int item) {
        if (head == null)
            head = tail = new ListNode(item, null);
        else {
            tail.next = new ListNode(item, null);
            tail = tail.next;
        }
    }

}
Package Visibility

- **public** declarations represent specifications—what clients of a package are supposed to rely on.
- package private declarations are part of the implementation of a class that must be known to other classes that assist in the implementation.
- **protected** declarations are part of the implementation that subtypes may need, but that clients of the subtypes generally won’t.
- **private** declarations are part of the implementation of a class that only that class needs.

<table>
<thead>
<tr>
<th></th>
<th>Same Package</th>
<th>Subclass</th>
<th>Everywhere</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>protected</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>default</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>private</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Next Time

• Head First Java
  – pp. 154-160, 587-591, 667-668