Lecture Set 5: Design and Classes

This Set:
- Basics of program design
- Pseudo code
- Objects and classes
- Heaps
- Garbage Collection
- More about Creating Objects and classes in Java
- Methods
- Constructors, Accessors, Mutators
- Equality
- Printing an object
- Unit testing

The Software Lifecycle ("waterfall")

The Software Lifecycle (actual)
In the Real World, Requirements and Design Rule

- Getting requirements right is essential for successful projects
  - FBI electronic case file (junked after $180m)
  - IRS system upgrade in late 90s (junked after >$2bn)
  - FAA air-traffic control (false starts, >$10bn spent)
- Good design makes other parts of lifecycle easier
- In "the real world" coding typically < 30% of total project costs
- A good design improves:
  - efficiency (speed)
  - efficiency (memory)
  - ease of coding
  - ease of debugging
  - ease of expansion

Usability Matters

Program Design

- There are many aspects to good design
  - Architecture
  - Modeling
  - Requirements decomposition
  - Pseudo-code
- In this class we will focus on latter
What Is “Pseudo-code”?

- When developing a complex part of a program (an algorithm), one of the tools often useful is pseudo-code.
- It’s not English, not programming language -- somewhere between.
- Captures the flow of the program without worrying about language-specific details.

Objects

- Bundles of (related)
  - data ("state")
  - operations ("behavior")
- Data often referred to as instance variables
- Operations usually called methods
- Invoking operations can change state (values stored in instance variables)

Sample Student Class

<table>
<thead>
<tr>
<th>State</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>getAge</td>
</tr>
<tr>
<td>ID</td>
<td></td>
</tr>
<tr>
<td>DOB</td>
<td></td>
</tr>
<tr>
<td>Major</td>
<td>getGrades</td>
</tr>
<tr>
<td>etc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>date → age</td>
</tr>
<tr>
<td></td>
<td>sem., class → grades</td>
</tr>
<tr>
<td></td>
<td>etc.</td>
</tr>
</tbody>
</table>
Sample Student Object

<table>
<thead>
<tr>
<th>Name</th>
<th>Kerry Keenan</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>444256895</td>
</tr>
<tr>
<td>DOB</td>
<td>04-22-1987</td>
</tr>
<tr>
<td>Major</td>
<td>CMSC</td>
</tr>
<tr>
<td>etc.</td>
<td></td>
</tr>
</tbody>
</table>

Methods
- getAge: date → age
- getGrades: sem., class → grades
- etc.

Accessing State / Methods

- If
  - o is an object
  - v is an instance variable of the object
  - m is a method of the object
- Then
  - o.v is how to access the data v in o
  - o.m() is how to invoke m in o
- So
  - If you have already done String str = "Jan"
  - str is an instance of an object!
  - Methods of this object: equals, compareTo, etc.
  - str.equals(), str.compareTo(), etc. invokes these methods on that object

Object-Oriented Programming

- Programs are collections of interacting objects
- Writing programs involves identifying what the objects should be and programming them
- Object-oriented languages provide features to ease object-oriented programming
- Defining objects involves indentifying
  - state
  - methods
Classes

- “Blueprints” ("templates") for objects
- Classes include specifications of
  - Instance variables (including types, etc.) to include in objects
  - Implementations of methods to include in objects
- Classes can include other information also, as will be seen later
  - static methods / instance variables
  - public / private methods, instance variables
  - And so on

Student Class Example

Conceptually:
- Instance variables:
  - String name
  - int ID
  - int dateOfBirth
  - String major
- Methods:
  - getAge()
  - getGrades()
  - etc.
- The actual class implementation will include code for the methods
- This describes a blueprint for student objects

How Are Objects Created?

- In Java: using new
  - Recall:
    - Scanner sc = new Scanner(System.in);
  - Invoking new:
    - creates fresh copies of instance variables in the "heap"
    - returns the “address” where the fresh variables are stored
  - Heap? Address?
Heap = “Fresh Memory”

- While a program is running, some memory is used to store variables
  - Terminology: stack
  - We have been representing the stack as a table, e.g.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>4.5</td>
</tr>
</tbody>
</table>

- Rest of memory is called heap and can be used for other purposes, including storing new objects

Main Memory

- Stack grows, shrinks during program execution (why?)
- So does “allocated heap” (part of heap in use)
- Unallocated part of heap is called “free”

Object Creation

- New space allocated in heap to store instance variables
  - Reference (= address) to this space is returned

Scanner sc = new ...
Strings Are Objects

- Where is `new` in
  - `String name = "Narita";`?
- Java provides it!
  - `String` is special because it is used so often
  - Java automatically “fills in” `new`
- You can too:
  - `String name = new String("Narita");`

In Java, 9 Sorts of Variables

- 8 primitive types
  - Types are the 8 built-ins (int, byte, double, etc.)
- Reference type
  - Objects always stored in heap (including all data)
  - Reference to objects are another type, and hold one memory address (typically one word)
- Stack holds local variables
  - e.g. `int x`
  - e.g. `String str;` // str is reference variable
- Heap holds allocated memory (i.e., with “new”)
  - e.g. `Scanner sc = new Scanner(System.in);`
  - e.g. `str = "Jan Plane";` // str is reference created above

Example

```java
int x = 7;
float y = 3.3;
String f = "cat";
```

```
Stack
- x = 7
- y = 3.3
- f

Heap
- x, y, f
```

```java
Scanner sc = new Scanner(System.in);
str = "Jan Plane";
```
Heap Issues

- What happens if \texttt{new} is called and there is no free heap? Crash!
- What happens if following are executed?

\begin{verbatim}
String s;
s = new String("cat");
s = new String("dog");
s = new String("cow");
\end{verbatim}

- Wasted heap
  - "cat", "dog" no longer referenced by stack
  - Crashes become a problem!

Garbage Collection

- This “heap management” or “memory management” issue is central in CS
- Java copes by invoking garbage collector to reclaim unused but still-allocated heap space
- Garbage collector reclaims memory in allocated heap and returns it to free heap
- In previous example, "cat" and "dog" would be reclaimed

Example

\begin{verbatim}
String a = new String("abc");
String b = new String("abc");
if (a == b) {
    println ("Equal");
} else {
    println ("Not equal");
}
\end{verbatim}

=> Not equal
Contrasting Example

```java
String a = new String("abc");
String b = a;
if (a == b) {
    println("Equal");
} else {
    println("Not equal");
}
=> Equal
```

- This is called **ALIASING**: Two variables refer to the same object.
- Can be DANGEROUS!!
- What if we really want to make a copy?
  ```java
  String a = "abc"
  String b = new String(a);
  ```

```

“equals”

- `==` checks if two reference variables refer to the same object
- Methods like `str.equals()` check if two different objects have the same “content”
- Other classes will have an `equals` method also

Classes in Java

- Class declarations have the following form in Java:

  ```java
  public class Student {
      // class body: instance variables, methods
  }
  ```

- When you create a class in Eclipse, it generates this template for you: 
Anatomy of an Instance Variable Declaration

```
public int IDNum;
```

Anatomy of a Method Declaration (1)

… for methods that do not return values

```
public void acceptTokens (int tokensPassedIn){
    tokenLevel = tokenLevel + tokensPassedIn;
    ...
}
```

Anatomy of a Method Declaration (2)

… for methods that return values

```
public int lastFour (){...
    return id % 10000;
}
```
Return Type

- Methods that return values must specify the type of the value to be returned
- The bodies of these methods use **return** to indicate when a value is to be returned
- The value being returned must have the same type as the return type

Object Creation

- Once a class is defined, objects based on that class can be created using **new**:
  
  ```java
  new Student();
  ```
- To assign an object to a variable, the variable’s type must be the class of the object
  
  ```java
  Student s = new Student();
  ```
- Each object has its own copies of all the instance variables in the class (except for certain kinds we'll study later)
- Instance variables and methods in an object can be accessed using `.` or using setter (mutator) methods
  
  ```java
  s.IDNum = 123456789;
  s.setIDNum(123456789);
  ```

Constructors

- Special "methods" in class definitions to specify how objects are created
- Form of a constructor definition:
  
  ```java
  Student (String nameDesired, int IDDesired, int tokensDesired) {
    name = nameDesired;
    id = IDDesired;
    tokenLevel = tokensDesired;
  }
  ```
- Can have more than one constructor, provided argument lists are different
  
  ```java
  Student (int IDDesired) {
    id = IDDesired;
  }
  ```
- Java includes default constructor (no arguments), which you can redefine (override)
  
  ```java
  Student () {
    tokenLevel = 3;
  }
  ```
Equality Testing

- Need to define what it means for two students to be equal

```java
public boolean equals(Student otherStudent) {
    if (otherStudent == null) {
        return false;
    } else if (id == otherStudent.id) {
        return true;
    } else {
        return false;
    }
}
```

Objects to Strings

- What happens if we try to print a Student object?
- **invoke println using a Student object as an argument?**
  - Student s1 = new Student();
  - System.out.println (s1);
- **Something like this prints:**
  - Student@82ba41

Java Knows “How” To Print Any Object

- Why?
  - Every class has a default toString method
  - toString converts objects into strings
  - System.out.println calls this method to print an object
  - Default: object type and address
- toString can be overridden!

```java
// The method for converting Students to strings
public String toString () {
    return (name + "": " + id);
}
```
Static Data Members and Static Methods

- Not contained in or associated with an object of that type
- Accessed by the `ClassName.variableName` or by `ClassName.methodName`
- rather than by `objectName.variableName` or by `objectName.methodName`

Set / Get Methods

- We have been using `=` to modify instance variables and accessing variables directly to read values
- Generally, this is not good practice because it imposes restrictions on class implementation
- Better
  - `set` methods to set values (mutators)
  - `get` methods to read values (accessors)

Set Methods (Mutators)

```java
public void setID(int newID) {
    id = newID;
}
```

- Can also do consistency checking

```java
public void setTokenLevel(int newTokenLevel) {
    if (newTokenLevel <= 3) {
        tokenLevel = newMonth;
    } else {
        System.out.println("Bad argument to setTokenLevel: "+ newTokenLevel);
    }
}
```
Get Methods (Accessors)

- Sole purpose is to return values of state
  ```java
  public int getID () {
    return id;
  }
  ```
- Why use them?
  - The state information may not always be stored in a single instance variable, since implementations may change
  - You give designers option of changing instance variables
  - Can log/monitor usage

Testing: The problem

- Problems:
  - need to be able to make sure all parts are tested
  - need to know in testing exactly which part was not as expected
  - need to be able to keep the tests for modifications made later
- Unit testing helps overcome this problems of making sure everything is tested
  - Unit testing: test each class and each part of the class (unit) individually
  - Goal is to eliminate inconsistencies between the API and the actual working of the code

Unit Testing

- Unit testing helps overcome this problems of making sure everything is tested in a structured way
  - Unit testing: test each unit individually (micro level – each method or specifically each interaction described in the API)
  - Goal is to eliminate errors within classes
- Needs for unit testing
  - Method for defining tests – inputs, expected outputs
  - Method for running tests
  - Method for reporting results
- One possibility: write a driver for each class
  - Driver class contains main method
  - main method creates objects in class to be tested, calls methods, prints outputs
  - User checks outputs, determines correctness
  - Good: easy, no special tools needed
  - Bad: tedious, relies on human inspection of outputs
- Another approach: JUnit
JUnit

- A unit-testing tool for Java
- Includes capabilities for:
  - Test definition, including output checking
  - Test running (execution)
  - Result reporting
- Seamless integration with Eclipse

Note
- In this class we will use JUnit 3.8.1
- So, when given a choice select JUnit 3

Structure of a JUnit 3.8.1 Test Case

```java
import junit.framework.TestCase;
public class FunnyIntegerSetTest01 extends TestCase {
  public void testInsert() {
    FunnyIntegerSet set = new FunnyIntegerSet();
    set.insert(3);
    assertTrue(set != null);
  }
  public void testFindClosest() {
    FunnyIntegerSet set = new FunnyIntegerSet();
    set.insert(3);
    set.insert(6);
    assertEquals(6, set.findClosest(5));
  }
}
```

A Test Case Is ... A Class!

- assertion checkers
  - `assertTrue(expression);`
    - If statement is true, keep running test; otherwise, halt test, report "fail"
  - `assertFalse(expression);`
    - If statement is false, keep running test; otherwise, halt test, report "fail"
  - `assertEquals(expression1, expression2);`
    - If expression1, expression2 equal, keep running test; otherwise, halt test, report "fail"
- If test terminates without failing an assertion and without throwing an uncaught exception, then it passes that test
- It continues with all subsequent tests regardless of passing or failing the current test
Hints on Testing

- Give names to tests that relate to class being tested
- Develop some tests before you code
  - Helps you to clarify what you are supposed to be doing
  - Gives you some ready-made tests to run while you code
- Use tests to debug
- How many tests?
  - **Statement coverage**: develop tests to make sure each statement in class is executed at least once (including constructors)
  - **Decision coverage**: develop tests to make each condition (if statement) in program both true and false
  - You should at least reach statement coverage in your own testing