Lecture Set 5: Design and Classes

This Set:
- Methods and Parameter Passing
- 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

The Software Lifecycle

- Requirements
- Design
- Coding
- Testing
- Deployment
- Maintenance
- Evolution

- New versions
- Bug fixes
- Delivery (documentation, etc.)

- What customers want
- What you plan to do
- Your program
- Did you meet requirements?
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

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 Object

<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>getGrades</td>
</tr>
<tr>
<td>DOB</td>
<td>date → age</td>
</tr>
<tr>
<td>Major</td>
<td>semester → grades</td>
</tr>
<tr>
<td></td>
<td>etc.</td>
</tr>
</tbody>
</table>

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**
  - System is an object, with out an instance variable
  - out is also an object, with println a method
  - System.out.println is how to access this method!
- **Suppose** str is a String
  - str is 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 identifying
  - 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:
  ```java
  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 stack as 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 primitively typed
  - Types are the 8 built-ins (int, byte, double, etc.)
  - Storage allocated on stack based on type
  - Value stored in stack
  - e.g. int x

- Reference typed
  - Types are classes
  - Storage allocated on stack to hold one memory address (typically, one word)
  - What is stored in stack is reference to heap, where actual data is stored
  - e.g. Scanner sc = new Scanner (System.in);

Example

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

- What happens if `new` is called and there is no free heap?
  Crash!
- What happens if following are executed?
  ```java
  String s;
  s = new String("cat");
  s = new String("dog");
  s = new String("cow");
  ```
- 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

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

- Not equal is printed

Contrasting Example

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

- Equal is printed
- This is called ALIASING: Two variables refer to 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
  }
  ```
  
  - Visibility modifier: more later in class

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

<table>
<thead>
<tr>
<th>Visibility modifier</th>
<th>Normal variable declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>public int</td>
<td>IDNum;</td>
</tr>
</tbody>
</table>

Anatomy of a Method Declaration (1)

... for methods that do not return values

<table>
<thead>
<tr>
<th>Visibility modifier</th>
<th>void keyword</th>
<th>method name</th>
<th>parameter list</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void</td>
<td>acceptTokens</td>
<td>(int tokensPassedIn)</td>
<td></td>
</tr>
</tbody>
</table>

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

body
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 (overide):
  ```java
  Student () {
      tokenLevel = 3;
  }
  ```
Equality Testing

- Need to defined 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` orby `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