Reading This Week

■ Chap 3

■ Reading for next week
  ■ re-read Chapter 4.1-4.6
News

- Assignment 1 due Tue Jan 31 5pm
- Extra TA hours in ICICS 008 to answer questions
  - Thu Jan 24 (today!) 4-6pm
    - Olivia Siu
  - Fri Jan 25 5-7pm
    - Ciaran Llachlan Leavitt
  - Sat Jan 26 12:30-2:30pm
    - Simon Hastings
- Weekly questions due today
  - Stay tuned for bboard postings with (some) answers
- Midterm reminder: Tue Feb 7, 18:30 - 20:00
  - Geography 100 & 200
Recap: Escape Characters

- How can we make a String that has quotes?
  - `String foo = "oh so cool";`
  - `String bar = "oh so \"cool\", more so";`

- Escape character: backslash
  - general principle
Recap: Random Numbers

- Random class in java.util package
  - public Random()
    - Constructor
  - public float nextFloat()
    - Returns random number between 0.0 (inclusive) and 1.0 (exclusive)
  - public int nextInt()
    - Returns random integer ranging over all possible int values
  - public int nextInt(int num)
    - Returns random integer in range 0 to (num-1)
Recap: Abstraction

- **Abstraction**: process whereby we
  - hide non-essential details
  - provide a view that is relevant
- Often want different layers of abstraction depending on what is relevant
Recap: Encapsulation and Info Hiding

Encapsulation: process whereby
- inner workings made inaccessible to protect them and maintain their integrity
- operations can be performed by user only through well-defined interface.
- aka information hiding

Hide fields from client programmer
- maintain their integrity
- allow us flexibility to change them without affecting code written by client programmer
  - Parnas' Law:
    - "Only what is hidden can by changed without risk."
Recap: Designing Classes

- Blueprint for constructing objects
  - build one blueprint
  - manufacture many instances from it

- Consider two viewpoints
  - client programmer: want to use object in program
    - what public methods do you need
  - designer: creator of class
    - what private fields do you need to store data
    - what other private methods do you need
Recap: UML

- UML diagram representing class design

<table>
<thead>
<tr>
<th>Classname</th>
</tr>
</thead>
<tbody>
<tr>
<td>- field: type</td>
</tr>
<tr>
<td>- method(): return type</td>
</tr>
<tr>
<td>+ Classname()</td>
</tr>
<tr>
<td>+ field: type</td>
</tr>
<tr>
<td>+ method(): return type</td>
</tr>
<tr>
<td>+ method(param1 type, param2 type): return type</td>
</tr>
</tbody>
</table>

private

public
Recap: UML

- UML diagram for `Die` class we designed

<table>
<thead>
<tr>
<th>Die</th>
</tr>
</thead>
<tbody>
<tr>
<td>- sides: int</td>
</tr>
<tr>
<td>+ Die()</td>
</tr>
<tr>
<td>+ setSides(numSides: int): void</td>
</tr>
<tr>
<td>+ roll(): int</td>
</tr>
</tbody>
</table>

private

public
Objectives

- understand how to design new classes using abstraction and encapsulation
- understand how to implement new classes in Java
Implementing Die

■ Last time
  ■ designed UML diagram
  ■ first draft of implementation
  ■ it compiled, but untested!

■ This time
  ■ refine implementation
  ■ test and debug implementation
Using Die

- Change hats from Die designer to Die user
- Roll two dice
  - print each value, and sum
- Design and implement RollDice driver: class with main method
public class RollDice
{
    public static void main ( String [] args)
    {
    }
Separation and Modularity

- Design possibilities
  - Die and RollDie as separate classes
  - one single class that does it all
- Separation allows code re-use through modularity
  - another software design principle
- One module for modeling a die: Die class
- Other modules can use die or dice
  - we wrote one, the RollDice class
- Modularization also occurs at file level
  - modules stored in different files
  - also makes re-use easier
Control Flow Between Modules

- So far, easy to understand control flow: order in which statements are executed
  - march down line by line through file
- Now consider control flow between modules

Client code

```java
int rollResult;
myDie.setSides();
rollResult = myDie.roll();
```

Die class methods

```java
public int roll()
{
    ...
}

public void setSides()
{
    ...
}
```
Designing Point: UML

- class to represent points in 2D space
Implementing Point

```java
public class Point {
}
```
Formal vs. Actual Parameters

- **formal parameter**: in declaration of class
- **actual parameter**: passed in when method is called
  - variable names may or may not match
- if parameter is primitive type
  - **call by value**: value of actual parameter copied into formal parameter when method is called
  - changes made to formal parameter inside method body will not be reflected in actual parameter value outside of method
- if parameter is object: covered later
Scope

- Fields of class are have **class scope**: accessible to any class member
  - in `Die` and `Point` class implementation, fields accessed by all class methods
- Parameters of method and any variables declared within body of method have **local scope**: accessible only to that method
  - not to any other part of your code
- In general, scope of a variable is block of code within which it is declared
  - **block** of code is defined by braces `{ }`
Key Topic Summary

Borrowed phrasing from Steve Wolfman

- Generalizing from something concrete
  - fancy name: abstraction

- Hiding the ugly guts from the outside
  - fancy name: encapsulation

- Not letting one part ruin the other part
  - fancy name: modularity

- Breaking down a problem
  - fancy name: functional decomposition