Designing and Implementing Classes

Topic 6

- Don't know much geography
- Don't know much trigonometry
- Don't know much about algebra
- Don't know what a slide rule is for

-Sam Cooke

Object Oriented Programming

- Object Oriented Programming is not new.
  - Smalltalk (1970s), Alan Kay's group at Xerox PARC
  - C++ (early 1980s), Bjarne Stroustrup, Bell Labs
  - Eiffel, Bertrand Meyer
  - Java and others

- Becoming the dominant style for implementing complex programs with large numbers of interacting components and in education

Main Tenets of OO Programming

- Encapsulation
  - abstraction, information hiding, responsibility driven programming
- Inheritance
  - code reuse, specialization
- Polymorphism
  - allows old code to use new code, do X, where X is different depending on the type of object

Tenants of Object Oriented Programming - Encapsulation

- information hiding, using something without knowing (or caring or having to worry) about how it works. abstraction

- achieved via classes and objects. Useful in breaking up large problems based on a study of the data involved first

- Don Knuth, Turing award winner
  algorithms + data structures = programming
Class and Objects

- **What are classes?**
  - *classes* are user defined data types. (as opposed to the primitive data types)
  - classes specify the operations that may be carried out on variables of this type. *Behaviors*
  - They also specify what sub data the variable consists of and have the implementation of the behaviors listed above.
  - *objects* are simply variables of a given class

- It is easier to describe how to use an object than it is to create a class
  - easier to teach some one to drive car or teach someone how to design and build a car?

Using Objects

- An *object* is a model (abstraction of) a real-world object.
  - it is a collection of variables (fields) and methods
- A *method* is an operation on an object.
- Objects interact with *messages*. A message is sent to an object and specifies the method that object should perform.
  - The sender (normally another object) doesn't know anything about how the method is performed

Using Objects

- Objects are variables, they have a declared type and a set of behaviors or operations
  
  String name1 = new String("Olivia A. Scott");
  String name2 = new String("Isabelle M. Scott");

- All object variables are reference variables (i.e. pointers)
- Objects must be dynamically allocated using the *new* operator and calling a constructor
  - special cases for Strings and arrays
- To determine the various behaviors of an object must look at the class documentation

The dot Operator

- Once an object is created the *dot operator* ( . ) is used to perform operations on the object or access sub data in the object:
  
  String name1 = new String("Olivia A. Scott");
  char first = name1.charAt(0);
  char second = name2.charAt(1);

- Any *public* part of the object may be accessed.
  - normally only *methods* are public.
  - Must call a method that actually exists and match one of the parameter signatures or syntax error
- the dot operator calls or invokes the method
- flow of control shifts to that code
**Object Data**

- Occasionally fields of an object are public, but this is generally considered bad style.
- Array length vs. String length
  ```java
  int[] list = new int[x];
  for(int i = 0; i < list.length; i++)
  ```
- Only a good idea if the data is constant.
- Good object oriented programming style calls for any changes to an object to be done via a method call.
- Normally a data’s object is hidden (encapsulated) and if I am using an object I don’t know or care about the details of that implementation.

**Object Methods**

- Object methods either
  - Tell you something about the object
    - *accessors*, have a return value
    - The charAt and length methods in String
  - Change the object in some way
    - *mutators*, no return value. Changes the state of the object
    - The setSize method in Rectangle
  - Do both, some people consider this bad style
    - *mutators that return a value*, return value and a change to the object’s state.

**Finding Object Methods**

  - Your best friend in the world
- [http://java.sun.com/j2se/1.4.1/docs/api/](http://java.sun.com/j2se/1.4.1/docs/api/)
- Has specification for all classes in the Java standard library
- Only documents the public portion, what we can use!
- A product of javadoc
  - Java source code in -> easy to read html out

**Classes**

- Fundamental unit of OO programming
- Java programs of any significance are a collection of classes

This is a rather smallish program and it contains 6 classes not counting another 6 classes from the Java Standard Library.
What are Classes?

- Classes are blueprints
  - They specify what new objects will contain and what can be done to the objects
  - Classes are analogous to data types and objects are the variables of that type

- Classes consist of an interface (what is it?) and an implementation (how it is done?)
  - Interface describes the ways an object of this type may be created via constructors and how a client may use or manipulate an object of this type via methods
  - Implementation is hidden and of no concern to the client. Implementation can be changed without affecting any client code

More on Classes

- Each class in Java is contained in its own file with the name matching the class name and with the .java extension
- The basics elements of a class are
  - Constructors
    - Used in the creation of objects. Similar to methods (procedures and functions).
    - Normally several overloaded for the convenience of the class users
  - Methods
    - Instance methods and class (static) methods
  - Variables (or fields)
    - Instance variables and class variables (static)
  - Constants
    - Instance (rare) and class (static)

The Die Class

- Consider a class used to model a die
- What is the interface? What actions should a die be able to perform?

- The methods or behaviors can be broken up into constructors, mutators, accessors

The Die Class Interface

- Constructors (used in creation of objects)
  - Default, single int parameter to specify the number of sides, int and boolean to determine if should roll
- Mutators (change state of objects)
  - Roll
- Accessors (do not change state of objects)
  - GetTopSide, GetNumSides, toString
- Public constants
  - DEFAULT_SIDES
Visibility Modifiers

- All parts of a class have visibility modifiers
  - Java keywords
  - public, protected, private, (no modifier means package access)
  - do not use these modifiers on local variables (syntax error)
- public means that constructor, method, or field may be accessed outside of the class.
  - part of the interface
  - constructors and methods are generally public
- private means that part of the class is hidden and inaccessible by code outside of the class
  - part of the implementation
  - data fields are generally private

The Die Class Implementation

- Implementation is made up of constructor code, method code, and private data members of the class.
- scope of data members / instance variables
  - private data members may be used in any of the constructors or methods of a class
- Implementation is hidden from users of a class and can be changed without changing the interface or affecting clients (other classes that use this class)
  - Example: Previous version of Die class, DieVersion1.java
- Once Die class completed can be used in anything requiring a Die or situation requiring random numbers between 1 and N
  - DieTester class. What does it do?

DieTester method

```java
public static void main(String[] args) {
    final int NUM_ROLLS = 50;
    final int TEN_SIDED = 10;
    Die d1 = new Die();
    Die d2 = new Die();
    Die d3 = new Die(TEN_SIDED);
    final int MAX_ROLL = d1.getNumSides() +
                        d2.getNumSides() + d3.getNumSides();

    for(int i = 0; i < NUM_ROLLS; i++)
        { d1.roll();
          d2.roll();
          System.out.println("d1: " + d1.getResult() +
                              " d2: " + d2.getTopSide() + " Total: " +
                              (d1.getTopSide() + d2.getTopSide()));
        }
```

DieTester continued

```java
    int total = 0;
    int numRolls = 0;
    do
        { d1.roll();
          d2.roll();
          d3.roll();
          total = d1.getTopSide() + d2.getTopSide()
                  + d3.getTopSide();
          numRolls++;
        }
    while(total != MAX_ROLL);
    System.out.println("nNumber of rolls to get " + MAX_ROLL + " was " + numRolls);
```
The Steps of Class Design

- Requirements
  - what is the problem to be solved
  - detailed requirements lead to specifications
- Nouns may be classes
- Verbs signal behavior and thus methods (also defines a classes responsibilities)
- walkthrough scenarios to find nouns and verbs
- implementing and testing of classes
- design rather than implementation is normally the hardest part
  - planning for reuse

Correctness Sidetrack

- When creating the public interface of a class give careful thought and consideration to the contract you are creating between yourself and users (other programmers) of your class
- Use preconditions to state what you assume to be true before a method is called
  - caller of the method is responsible for making sure these are true
- Use postconditions to state what you guarantee to be true after the method is done if the preconditions are met
  - implementer of the method is responsible for making sure these are true

Precondition and Postcondition Example

```java
/* pre: sides > 1
   post: getTopSide() = 1, getNumSides() = sides
*/
public Die(int numSides)
{
    if(numSides <= 1)
    {
        throw new IllegalArgumentException("numSides is " +
        numSides + ". numSides must be > 1");
    }
    iMyNumSides = numSides;
    iMyResult = 1;
    assert getTopSide() == 1 & getNumSides() == numSides;
}
```

Object Behavior - Instantiation

- Consider the DieTester class
  ```java
  Die d1 = new Die();
  Die d2 = new Die();
  Die d3 = new Die(10);
  ```
- When the new operator is invoked control is transferred to the Die class and the specified constructor is executed, based on parameter matching
- Space(memory) is set aside for the new object's fields
- The memory address of the new object is passed back and stored in the object variable (pointer)
- After creating the object, methods may be called on it.
Creating Dice Objects

Objects

- Every Die object created has its own instance of the variables declared in the class blueprint
  ```java
  private int iMySides;
  private int iMyResult;
  ```
- thus the term *instance variable*
- the instance vars are part of the hidden implementation and may be of *any* data type
  - unless they are public, which is almost always a bad idea if you follow the tenets of information hiding and encapsulation

Complex Objects

- What if one of the instance variables is itself an object?
- add to the Die class
  ```java
  private String myName;
  ```

The Implicit Parameter

- Consider this code from the Die class
  ```java
  public void roll()
  { 
    iMyResult = ourRandomNumGen.nextInt(iMySides) + 1;
  }
  ```
- Taken in isolation this code is rather confusing.
- what is this iMyResult thing?
  - It's not a parameter or local variable
  - why does it exist?
  - *it belongs to the Die object that called this method*
  - if there are numerous Die objects in existence
  - Which one is used depends on which object called the method.
The *this* Keyword

- When a method is called it may be necessary for the calling object to be able to refer to itself – most likely so it can pass itself somewhere as a parameter
- When an object calls a method an implicit reference is assigned to the calling object
- The name of this implicit reference is *this*
- *this* is a reference to the current calling object and may be used as an object variable (may not declare it)

```java
// in some class other than Die
Die d3 = new Die();
d3.roll();

// in the Die class
public void roll(){   
    iMyResult = ourRandomNumGen.nextInt(iMySides) + 1;
    /* OR
    this.iMyResult...
    */
}
```

A Possible Equals Method

```java
public boolean equals(Object otherObject)
{
    Die other = (Die)otherObject;
    return iMySides == other.iMySides && iMyResult == other.iMyResult && myName.equals(other.myName);
}
```

- Declared Type of Parameter is Object not Die
- Override (replace) the equals method instead of overload (present an alternate version) – easier to create generic code
- We will see the equals method is *inherited* from the Object class
- Access to another object's private instance variables?

An equals method

- Working with objects of the same type in a class can be confusing
- Write an equals method for the Die class. Assume every Die has a myName instance variable as well as iMyNumber and iMySides
Another equals Methods

```java
public boolean equals(Object otherObject)
{
    Die other = (Die)otherObject;
    return this.iMySides == other.iMySides
        && this.iMyNumber == other.iMyNumber
        && this.myName.equals( other.myName );
}
```

Using the this keyword / reference to access the implicit parameters instance variables is unnecessary. If a method within the same class is called within a method, the original calling object is still the calling object.

A "Perfect" Equals Method

From Cay Horstmann's *Core Java*

```java
public boolean equals(Object otherObject)
{
    // check if objects identical
    if( this == otherObject)
        return true;
    // must return false if explicit parameter null
    if(otherObject == null)
        return false;
    // if objects not of same type they cannot be equal
    if(getClass() != otherObject.getClass() )
        return false;
    // we know otherObject is a non null Die
    Die other = (Die)otherObject;
    return iMySides == other.iMySides
        && iMyNumber == other.iMyNumber
        && myName.equals( other.myName );
}
```

the instanceof Operator

- `instanceof` is a Java keyword.
- part of a boolean statement

```java
public boolean equals(Object otherObj)
{
    if otherObj instanceof Die
    {
        //now go and cast
        // rest of equals method
    }
}
```

- Should not use `instanceof` in equals methods.
- `instanceof` has its uses but not in equals because of the contract of the equals method.

Class Variables and Class Methods

- Sometimes every object of a class does not need its own copy of a variable or constant
- The keyword `static` is used to specify class variables, constants, and methods

```java
private static Random ourRandNumGen = new Random();
public static final int DEFAULT_SIDES = 6;
```

- The most prevalent use of static is for class constants.
  - if the value can't be changed why should every object have a copy of this non changing value
Class Variables and Constants

All objects of type Die have access to the class variables and constants.

A public class variable or constant may be referred to via the class name.

Syntax for Accessing Class Variables

```java
public class UseDieStatic
{
    public static void main(String[] args)
    {
        System.out.println( "Die.DEFAULT_SIDES " + Die.DEFAULT_SIDES );
        // Any attempt to access Die.ourRandNumGen
        // would generate a syntax error
        Die d1 = new Die(10);
        System.out.println( "Die.DEFAULT_SIDES " + Die.DEFAULT_SIDES );
        System.out.println( "d1.DEFAULT_SIDES " + d1.DEFAULT_SIDES );
        // regardless of the number of Die objects in
        // existence, there is only one copy of DEFAULT_SIDES
        // in the Die class
    }
    // end of main method
}
// end of UseDieStatic class
```

Static Methods

- static has a somewhat different meaning when used in a method declaration
- static methods may not manipulate any instance variables
- in non static methods, some object invokes the method `d3.roll();`
- the object that makes the method call is an implicit parameter to the method

Static Methods Continued

- Since there is no implicit object parameter sent to the static method it does not have access to a copy of any objects instance variables
  - unless of course that object is sent as an explicit parameter
- Static methods are normally utility methods or used to manipulate static variables (class variables)
- The Math and System classes are nothing but static methods
static and this

Why does this work (added to Die class)

```java
public class Die {
    public void outputSelf() {
        System.out.println( this );
    }
}
```

but this doesn't?

```java
public class StaticThis {
    public static void main(String[] args) {
        System.out.println( this );
    }
}
```