BJP Chapter 8: Classes

A big part of CSC 20 is practicing using class definitions to create new variable types. Why? As the programs you write become larger, the main technique used to combat complexity is modularity -- implementing logical pieces of the program independently. This happens naturally when you design and create new variable types. Modular programming promotes two important properties: abstraction and encapsulation.

Abstraction: separating (complex) implementations from (simple) interfaces.

Radios, cars, ovens, etc are all complex devices with simple controls. For many, a radio "is" just three things: on/off switch, volume knob, tuning knob. This abstraction greatly simplifies the use of the radio. The user of the radio does not need to know how it works internally. As long as the radio works as expected, the user is happy.

Encapsulation: Bundling together implementation details for an abstraction and restricting access to them (only the abstraction is accessible to users).

A radio encapsulates the electronics that make it work. The radio controls expose the radio's abstraction to the world.

Modular Programming: Using abstraction and encapsulation to break big programs into smaller pieces.

Class writing process:
1. Decide what thing you want to represent as a new variable type.
2. Decide what methods each object of this type should support.
3. Decide what data each object of this type needs to remember.
4. Implement, test, debug, document and refine.

Good Practice: As you develop your classes you should do the following.
- Test: Each time you write something significant, write some simple code that verifies that it works as expected. Don't continue until you're satisfied. By testing frequently, you catch problems early, making them easier to fix.
- Follow coding conventions: Follow the textbooks example for style (ie, whitespace, naming, etc). Or better yet, follow the Google Java Style Guide.
- Document implementations: Any code that is non-obvious should have inline comments describing what is going on. This helps people maintain the code at a later date.

- Document interfaces: Javadoc is a simple tool that generates html documentation describing how someone uses your class. It should be comprehensive in that it should fully explain the class's requirements and how it behaves whether the requirements are satisfied or not.
- Make robust: Clients of a class should not be able to make it behave other than as documented.

Example 1: We want to write programs that manipulate dice, so we'd like to be able to declare variables of type `Die`.

```java
import java.util.Random;

/**
 * A class representing a single Die. Upon creation, the user must pass to the
 * constructor the number of sides on the Die. Each Die object keeps track of
 * both the number of sides the Die has, and the current value of the Die.
 * Possible values for a Die are 1 through to the number of sides. Subsequent
 * calls to a Die's roll method cause the Die to take on a new random value,
 * which can be accessed through either the getValue or toString methods.
 *
 * @author Ted Krovetz
 * @version 2 September 2015
 */
public class Die {

    private int value;
    private int sides;
    private Random rand;

    /**
     * Initializes a new Die with its number of sides. The die can then take
     * on any value 1..sides. An initial random roll takes place too.
     *
     * @throws IllegalArgumentException if sides is not greater than 0.
     * @param sides sets the number of sides on the Die. The sides have values
     *              1 through sides.
     */
    public Die(int sides) {
        if (sides < 1) {
            throw new IllegalArgumentException("sides must be positive");
        }
        this.sides = sides;
        rand = new Random();
        value = rand.nextInt(sides) + 1;
    }

    /**
     * Sets the Die's value to a new randomly chosen one. The value will be in the
     * range 1 through the number of sides on the Die.
     */
    public void roll() {
        value = rand.nextInt(sides) + 1;
    }

    /**
     * Returns the current value of the Die as an integer.
     *
     * @return the current integer value of the Die.
     */
    public int getValue() {
        return value;
    }

    /**
     * Returns the current value of the Die as a String
     * (eg, value 2 is returned as "2").
     *
     * @return a String representation of the Die's current value.
     */
    public String toString() {
        return ""+value;       // Trick to produce a String from an int
    }

    // Simple embedded test.
    public static void main(String[] args) {
        try {
            Die doesntWork = new Die(0);
            System.out.println("Shouldn't get here!");
        }
        catch (IllegalArgumentException e) {
            // Do nothing -- this exception is expected
        }
        Die d1 = new Die(6);
        Die d2 = new Die(6);
        for (int i=0; i<10; i++) {
            System.out.println("Die 1: " + d1 + ", Die 2: " + d2 + ".");
            d1.roll();
            d2.roll();
        }
    }
}
```

Important things to remember:

- Make fields and helper methods private.
- Initialize non-static fields in your constructor.
- Test as you go: write something simple you can test, test it, repeat; until you're done.

Example 2: A program that wants to imitate a Magic 8 Ball might want to declare a variable to represent it and then manipulate it through methods.

```java
import java.util.Random;

/** The Magic8Ball class simulates the properties of a classic Magic 8 Ball
 *  toy, designed by Mattel in the 1950s. With the original toy, a user
 *  would ask a yes/no question, shake the ball, and read the answer from
 *  a window in the ball. This class offers a "shake" method which randomly
 *  selects from 20 answers. The client can then invoke the class's "toString"
 *  method to retrieve the answer.
 *
 *  @author Ted Krovetz (tdk@csus.edu)
 *  @version 12 September 2016
 */
public class Magic8Ball {

    private static Random rand;
    private String curReply;
    private static String[] replies = {
        "It is certain",
        "It is decidedly so",
        "Without a doubt",
        "Yes, definitely",
        "You may rely on it",
        "As I see it, yes",
        "Most likely",
        "Outlook good",
        "Yes",
        "Signs point to yes",
        "Reply hazy try again",
        "Ask again later",
        "Better not tell you now",
        "Cannot predict now",
        "Concentrate and ask again",
        "Don't count on it",
        "My reply is no",
        "My sources say no",
        "Outlook not so good",
        "Very doubtful"
    };

    /** Initializes object's state, including to an initial random reply. */
    public Magic8Ball() {
        rand = new Random();
        shake();
    }

    /** Updates object to have a new randomly chosen reply. */
    public void shake() {
        int idx = rand.nextInt(replies.length);
        curReply = replies[idx];
    }

    /** Returns the current object's answer as a string. The answer is and
     *  appropriate one for a yes/no question.
     *
     * @return current object's answer
     */
    public String toString() {
        return curReply;
    }

}
```

Static: These past two examples have something in common. They both make every single object have their own Random object, and this is inefficient. It would be much better if there was just one Random object that every Die object or Magic8Ball object shared. This is what the `static` keyword is for.

If you declare a method or field `static`, then you are telling java that it should not place the method or field in every object, and instead there should be just one that every object shares. By defining the field `private static Random rand` instead of `private Random rand`, we're saying that there should be a single variable `rand` that every Die object shares.

Rules:
~ non-static methods can access both static and non-static methods and fields.
~ static methods can only access static methods and fields.
~ static fields and methods are usually accessed using the class name to make it clear it's a static access (eg, `Die.rand` is the static `rand` found in the `Die` class)
~ Do not initialize static fields in a constructor (or do so carefully), constructors are per-object.

```java
public class Die {

    private int value;
    private int sides;
    private static Random rand;

    public static final int DEFAULT_DIE_SIDES = 6;

    static {
        rand = new Random();
    }

    public Die(int sides) {
        if (sides < 1) {
            throw new IllegalArgumentException("sides must be positive");
        }
        this.sides = sides;
        value = rand.nextInt(sides) + 1;
    }
}
```

Fields declared static are often called "class variables" because there is one of them per class, whereas non-static fields are often called "instance variables" because there is one per object instance.

Javadoc: It's been said that over half of the time a professional programmer spends with code, it is with code originally written by someone else. This means that when you write code, you should keep this reality in mind, and write your code to be easily understood by someone else. In general, this means:

- Follow standard formatting conventions (ie, capitalization, indentation, etc).
- Give methods and variables meaningful names.
- Write code simply rather than cleverly.
- Write inline comments describing any non-obvious code.
- Write documentation describing the purpose and function of every class and public method.

In Java, this last step is done using the `javadoc` tool. If you've ever looked at the Java documentation for any of the built-in classes (String, Random, Scanner, etc) then you've seen what javadoc output looks like. It's remarkably easy to produce.

To learn how to write javadoc:

1. Read Appendix B from our textbook.
2. Look at some examples. Google "javadoc string" and "javadoc random" to see what kind of information is in a thorough javadoc. We won't be as thorough in this class, but seeing Java's own documentation exposes you to what a really good javadoc looks like.
3. Read some tutorials. Google "javadoc tutorial" and look over the top couple of hits. You'll see examples and descriptions of the various parts.

What's required in this course? The rule-of-thumb is to write as much as would be required to satisfy *you* if you were a client of the software. Really! Ask yourself if you'd be happy with this level of documentation of a class you were trying to use.

- Include class documentation (just before the `public class Foo`). This should explain at a high-level the purpose of the class and how it's used. If there is something interesting or important about how it is implemented, this can be discussed here. Look at some good examples of Javadoc to see what kinds of things are mentioned.
- Every public method (except any embedded test `main`) should have an entry describing the purpose of the method and how it's used. Again, if there is something interesting or important about how it is implemented, this can be discussed here. Look at some good examples of Javadoc to see what kinds of things are mentioned.
- Use @author, @version, @param, @return, and @throws tags every chance you get.