Review of Lab3

1. A Map for Coupon values

In the Coupon class we want to have a constructor that accepts a Color, so that the coupon class knows the value of colors.

Coupon coupon = new Coupon("red");

The simplest way is to write code like this:

```
public class Coupon implements Valuable {
   private String color;
   private double value;
   public Coupon(String color) {
      color = color.toLowerCase();
      if ( color.equals("red") ) this.value = 100;
      else if ( color.equals("blue") ) this.value = 50;
      else if ( color.equals("green") ) this.value = 20;
      else
        throw new IllegalArgumentException("Invalid color "+color);
      this.color = color;
   }
```

The code is messy and gets worse as we add more colors. In Java 7, you can use a **`switch`** statement with String, but its still messy.

How To Use a Map

A Map is a collection of key-value pairs. The keys are stored in a way that lets the map quickly get the value for any key. The keys and values can be any reference type you like (not primitives). The keys should have valid equals and hashCode methods, because Map calls these methods to locate key values.

A Map requires 2 type parameters: the type of the **Key** and the type of the **Value**. For example, to create a Map from String to Integer (useful for counting strings) you'd write:

```
Map<String, Integer> map = new HashMap<String, Integer>( );
```

`Map` (and **`HashMap`**) has these methods (and many more – see the Java API):

`void put(K key, V value)`	add `(key,value)` to the map
`V get(Object key)`	get the value of a `key`. Returns `null` if the `key` is not found in map.
`boolean containsKey(K key)`	return true if `key` is in the map, false otherwise
`V remove(K key)`	remove a key and its value. Returns the value of key.
`Set<K> keySet()`	get all the keys in the map.

To add colors and values to a map:

```
map.put("red", 100.0);
map.put("blue", 50.0);
// Taksin's favorite color...
map.put("gold", 1000.0);
```

Java automatically converts the primitive values (100.0) to wrapper objects (Double(100.0)). This is called ***auto-boxing***.

1.1 How to Initialize a static Map?

The Coupon class only needs *one* map for color values; it would be silly to have a separate Map for each Coupon. So we should declare the map to be **`static`**. And we need to put the data in the map **before** anyone starts creating Coupon objects.

We can't put values in the Map using its constructor – we need to write some code. Where can we put the code so that it is executed before the class is used? Java allows you to create a **static block** that is executed with the class is loaded into memory. **Static block** is like a constructor for the class – it initializes static values. The syntax is:

```
   // a "static block" is executed when class is first loaded
   static {
      // put your static code here...
      System.out.println("initializing Coupon class");
   }
```

So let's use a *static block* to create and initialize the Map:

```
import java.util.Map;     // Map interface
import java.util.HashMap; // Map implementation

public class Coupon implements Valuable {
   private static Map<String,Integer> colors;

   // a "static block" is executed when class is first loaded
   static {
      colors = new HashMap<String,Integer>( );
      colors.put("red", 100);
      colors.put("blue", 50);
      ...
   }

   /** coupon constructor */
   public Coupon(Color color) {
      //TODO use the map to get the value of this coupon color
```

2. An Enum for Coupon Types

Using a Map for coupon values is good practice using Map, but the colors and values are buried in the Coupon class. And, there is nothing to prevent someone from trying to create a coupon with invalid color, like: new Coupon("Brown").

For type safety and to separate the coupon *types* from the Coupon class, we could use an Enum. An Enum is a fixed collection of named values, which usually represents a programmer-defined type. For example, an enum for 4 directions:

```
public enum Direction {
    NORTH,
    SOUTH,
    EAST,
    WEST;  // final semi-colon is optional here
}
```

Use enum like a class with public static values that are objects of that enum type. For example:

```
    Direction dir = Direction.NORTH;
```

The members of an enum can also have attributes and methods. We can use this to associate a value with each coupon color:

```
public enum CouponType {
    RED(100),
    BLUE(50),
    GREEN(10);
    // attributes of the enum
    public final double value;
    // enum can have a constructor, but it must be private
    private CouponType(double value) { this.value = value; }
}
```

The **value** is defined as public so we can access it directly by writing `RED.value` and **final** so it cannot be changed. You could declare **value** as **private** and provide a getValue() method.

Enum has some automatically generated methods: CouponType.values() returns an array of all the enum members, CouponType.valueOf("RED") returns the enum member Coupontype.RED. CouponType.RED.name() returns "RED" (String name of the enum member). See the Java API docs for "Enum" for other enum methods.

The benefit using an enum is *type safety*. If the Coupon constructor requires an enum value (rather than a String), then the compiler can verify that a valid coupon type has been entered.

```
public class Coupon {
    public Coupon( CouponType ctype ) {
        // set the value of this coupon, or save ctype as attribute
    }
```

References to learn more about enum are:

- Java Tutorial has a section on enum
- Java API doc for Enum (the superclass of all enum)
- OOP lecture slides on enum

3. Coupon Factory

The third way to create coupons for different colors is to define a *Factory Method* in another class. The separates the task of *creating objects* from the task of *using* objects. Factories are often used when object creation is complex, or you want to control how objects are created. An example of a simple factory method is in the Calendar class:

```
Calendar today = Calendar.getInstance();
```

we will study factory methods and the *Factory Method* design pattern in the course (OOP2).