Object Oriented Programming with Java

Enumerations

Enumeration – Rationale

You need a variable to represent a day of the week. It can obviously have 7 values **only**. What data type are you going to use?

String... char… int… byte?

Is this “safe” from a programming perspective?

How can you use one of the reserved keywords we learned to make this more “safe”?

The **`enum`** type was created for this reason

Enumeration – Purpose & Implementation

Purpose: list out all of the values in a **finite** set

- examples: days of week; planets; grades
- more error-proof than just using ints or Strings: less chances to abuse the value

Implementation:

- Java uses the class mechanism behind the scenes
- an enumeration is "syntactic sugar" for a special usage of classes

Enumeration – Creation

```
public enum Grade
{
    A, B, C, D, F
}
```

- Place enumeration in its own file, like a class (above: Grade.java)
- An enumeration creates a new **type**, like classes and interfaces
- The values of enumeration are… enumerated (i.e. listed explicitly)
- It looks like set notation in math!

Enumeration – Usage

- Direct usage of the values: **`EnumName.EnumValue`**
  Example: **`Grade.A Grade.C Grade.F`**

- switch usage is allowed; here, don't use **`enumName`**: just value.

```
//directly access the enum fields
Grade g = Grade.A;
System.out.println("The grade is " + g);

//notice that we DON'T say Grade.A, just A, inside a switch:
switch (g) {
case A:
    System.out.println("Ace!!!!");
    break;
case B:
    System.out.println("Buzz...");
     break;
default:
    System.out.println("Meh....");
}
```

Enumeration – Iteration

Use the **`values()`** method provided for every enumeration type to get a listing of all values in an enumeration, and use a *foreach* loop:

```java
for (Grade g : Grade.values())
{
    System.out.println("Grade: " + g);
}
```

Enumeration – Cheatsheet

- When to use enums? Any time you need a fixed set of **constants**, whose values are known at **compile-time**
- It is not necessary that the set of constants in an enum type stays fixed for all time. You can add new constants to an enum without breaking the existing codes.
- Enums are type-safe!
- All constants defined in an enum are public static final and accessed via **`EnumName.instanceName`**
- You do not instantiate an enum
- Enums can be used in a switch-case statement, just like an int.
- Whenever an enum is defined, a class that extends **`java.lang.Enum`** is created. Hence, enum cannot extend another class or enum.

Enumeration – Built-in Methods

The `java.lang.Enum` has the following built-in methods:

`public final String name()`
Returns the name of this enumeration constant, exactly as declared in its enum declaration

`public String toString()`
You can override the `toString()` to provide a customized description

`public final int ordinal()`
Returns the ordinal of this enumeration constant

Enumeration – Behind the scenes

The compiler creates an instance of the class for each constant defined inside the enum.

Consider the following enum:

```
public enum Size {
   BIG, MEDIUM, SMALL
}
```

Let's decompile this enum and see what the compiler generates:

```
> javac Size.java

> javap -c Size.class
Compiled from "Size.java"
public final class Size extends java.lang.Enum {
  public static final Size BIG;
  public static final Size MEDIUM;
  public static final Size SMALL;
  public static Size[] values();
  ......
  public static Size valueOf(java.lang.String);
  ......
  static {};
  ......
```

Conclusion:

A enum extends from **`java.lang.Enum`** and all its values are defined as **`public static final`**

Practice Problems

1. Create an enumeration for the sign of a number (positive, negative, zero)

2. Create a variable that uses your **`Sign`** enumeration. Use it in a switch to print out if the number is "bigger than", "equal to" , or "less than" zero.

3. Create an enumeration, **`Quadrant`**, that represents the four quadrants of the two-dimensional plane.

Advanced Enumerations

- adding fields
- adding private constructors

```java
public enum Day {
    // we add special constructor calls to the enumerated values.
    MON ("Monday",false),  TUES("Tuesday",false),  WED("Wednesday",false),
    THURS("Thursday",false),  FRI("Friday",false),  SAT("Saturday",true),
    SUN("Sunday",true);

    //we can create fields. (private just for encapsulation reasons)
    private String fullDesc;
    private boolean isWeekend;

    //(private/package-private only) constructor, called above.
    private Day (String fullDesc, boolean isWeekend){
        this.fullDesc = fullDesc;
        this.isWeekend = isWeekend;
    }
}
```

Advanced Enumerations

Adding other methods

- At this point, our enumeration can be roughly treated like the class that it implements (a child class of the Enum class, with special "syntactic sugar" and behavior enforced).
- Adding methods that can use the non-public things in the definition is just like adding methods to a class with similar fields/constructors

```java
// being implemented as a class, we can also provide toString()
public String toString()
{
  return "<Day: "+fullDesc+" is"+(isWeekend?"n't":"")+" weekday>";
}

//other public methods are also allowed
public String other(int n)
{
   return "Other "+fullDesc+" description..."+n;
}
```

Practice Problems

4. Update your `Quadrant` enumeration to have two private fields for the `x` and `y` Signs. Bonus complexity: use your `Sign` enumeration for these!

5. Add constructor calls to use these fields for your `Quadrant` values.

6. Write a `toString` method for your `Quadrant` enumeration.

7. Write two methods: `getXSign` and `getYSign`. They work on a `Quadrant` value (no parameters), returning the `Sign` value for an x/y value in this quadrant.