

Chapter 9

Enumerations

The Need for Something New

When writing programs, we occasionally come across a small set of values that we need to represent. Some examples:

- the days of the week
- the model# of car
- the color of a traffic light
- the flags (options) that may be enabled or disabled for our program.

Although we could just do something simple, such as choose integers for each value:

```
public final int RED = 0;
public final int YELLOW = 1;
public final int GREEN = 2;
```

There's nothing to stop us from saying RED+1, or get mistakes such as this:

```
myColor = 5;
if (myColor == RED) {
    car1.waitForGreen();
}
else if (myColor == YELLOW) {
    car1.clearIntersection();
}
else //assuming green, but we have a non-color!
{
    car1 . enterIntersection();
}
```

We might choose instead to use Strings, like "red", "green", "yellow". We still have no control over the values: what if we end up with "Red"? Or what about String values like "pink", "", "gray" vs "grey", and so on? Although we've chosen a type that has enough values for us to stand in for each of our values, we are forced to have more values than we want to deal with, and we have to allow all the operators that already exist (over ints or String, for instance).

We want a way to explicitly list the values of our new type, rather than re-use another type as a proxy for the values we actually have in mind. We want to **enumerate** the values.

The Answer: Enumerations

An enumeration defines a new type (much like a class definition), and the enumeration definition explicitly lists all the values of that type:

```

public enum Day { SUN, MON, TUES, WED, THURS, FRI, SAT }

public enum Simpson { BART, LISA, MAGGIE, HOMER, MARGE }

public enum Digit { THUMB, INDEX, MIDDLE, RING, PINKY }

public enum Ghost { BLINKY, PINKY, INKY, CLYDE }

public enum MyBool { TRUE, FALSE }

```

Because the values are unchanging representations of one value, we name them with the same convention as final constants: ALL_CAPS_WORDS_SEPARATED_BY_UNDERSCORES.

Just like classes, we place an enumeration in a file by itself, named as *<enumIdentifier>.java*. For example, we would have *Day.java*, *Simpson.java*, *Digit.java*, *Ghost.java*, and *MyBool.java* above.

Just like classes, we can use an enumeration any time it is visible; placing the enumeration in the same folder means that it will be in the same package, and thus available in other classes in the package. We could also state that the enumeration exists in a specific package, and then import the enumeration through a package import statement.

Usage

Since an enumeration is a type, we can create variables that store one value of the type. We refer to the values in a fashion similar to static variables:

```

EnumerationName . VALUE_NAME

public class TestEnums {
    public static void main(String[] args){
        Day day = Day . SUN;
        Simpson simp = Simpson . BART;
        Digit finger = Digit . PINKY;
        Ghost ghost = Ghost . PINKY;

        System.out.println(day+"\n"
            + simp+"\n"
            + finger+"\n"
            + ghost
        );
    }
}

```

We can use switch statements with enumerations (as of Java 1.7). Because the expression we're matching indicates the enumeration type that we are using, and each case must have a value of that type, we do not state the qualified name (*Ghost.BLINKY*), just the value (*BLINKY*).

```

String style= "";
switch (ghost) {
    case BLINKY:
        style = "chaser";
        break;
    case PINKY:
        style = "ambusher";
        break;
    case INKY:

```

```

        style = "fickle";
        break;
    case CLYDE:
        style = "stupid";
    }
    System.out.println("ghost: " + ghost + "; style = " + style);
}
}

```

Your Turn!

- Create an enumeration from any of the following:
 - characters in Super Mario Bros.
 - the five stages of grief (denial, anger, bargaining, depression, acceptance). (Formal name: the Kübler-Ross Model).
 - types of coins (penny, nickel, dime, quarter...)
 - make up your own enumeration! 😊
- In another file, create a variable (or many) that can hold a value of your enumeration type. Store one of the enumeration's values into it, and print it.
- Use your enumeration in a switch statement.
- Use your enumeration in an if-statement, checking e.g. `if (day==Day.MON)`. Does this work, or will you need to rely on the `.equals()` method? (Is `.equals()` even available? Try it out! That was defined for classes via the `Object` class being the ancestor of all classes, but this is an enumeration...).

Enriching Our Enumerations

Java's enumerations are actually much more useful than just listing out values. It turns out that enumerations are implemented as a special, restricted usage of classes. Your enumeration implicitly inherits from `java.lang.Enum`, which is a class.

So, what class-like features can we add to an enumeration definition?

- fields
- constructors (private/default-visibility only)
- methods

Fields and Constructors in Enumerators

In the above example, we had to use a `String` to represent the style of attack for each Pac-Man ghost. It would be nice if each value in the `Ghost` enumeration had an associated `String` value to describe its attack style.

We will add a field to the `Ghost` enumeration definition, so that each ghost has its own movement style. However, we will still only be able to access the values through `EnumName.VALUE_NAME` as before.

Because the values are supposed to be unchanging and fully listed out, though, we can't call a constructor to create them – there is exactly one of each value, always. Instead, we will add a private constructor to deal with setting up our newly added field, and we will also add constructor calls for each value. By adding all three things at once (field, private constructor, constructor calls for each value), we achieve the desired effect:

```

public enum Ghost {
    /* we add in parameters to call the constructor.
       These look odd as constructor calls without using
       the Ghost identifier, but that's what they are.
    */
    BLINKY("Blinky", "attacker") ,
    PINKY ("Pinky", "ambusher") ,
    INKY  ("Inky", "fickle") ,
    CLYDE ("Clyde", "stupid");

    // all 4 ghosts have their own movementStyle value.
    private final String name;
    private String movementStyle;

    //constructor: set up the field(s) for each Ghost value.
    private Ghost (String name, String movementStyle) {
        this.name = name;
        this.movementStyle = movementStyle;
    }
}

```

We can also add methods to our enumeration definition. The methods may be any visibility, and may or may not be static.

```

    //public methods are available.

    //getter (but no setter) for private final field.
    public String getName(){ return name; }

    //getters and setters for non-final field:
    public String getMovementStyle(){ return movementStyle;}
    public void setMovementStyle(String newval) {
        movementStyle=newval;
    }

    //because we implicitly inherit from java.lang.Enum,
    //we have Object methods available to override:
    public String toString() {
        return name+"(" +movementStyle+")";
    }
}

```

We can now add to our testing code (TestEnums.java).

```

Ghost g1 = Ghost.BLINKY;    //no constructor call - just name.
System.out.println("g1 is named " + g1.getName());
System.out.println("g1 toString: " + g1);
g1.setMovementStyle("stationary");
System.out.println("g1 toString: " + g1);

```

We can also use an enumeration as an Iterator, by getting an array of all the values. The static method `values()` returns an array containing all the enumeration's values. It's as if the following definition was provided explicitly in our enumeration definition:

```

// this values definition is automatically made for you.
public static Ghost[] values () {
    Ghost[] ghosts = {BLINKY, PINKY, INKY, CLYDE};
    return ghosts;
}

```

```
}
```

Indeed, the Java compiler adds this definition to your enum definitions every time.

Let's test out our code:

```
System.out.println("Testing for-each Ghost.values() :");  
for (Ghost gval : Ghost.values() ) {  
    System.out.println(gval);  
}
```

Your Turn!

- Add a private field to your enumeration.
- Add a constructor method (or many, via overloading) that will set that field.
- Add a parameters list to each listed value to use a constructor method definition that you just added.
 - Make sure your code still compiles!
- Add a getter method for your field.
- Add more testing code that now uses your getter method to see that the enumeration values do indeed have their own private field values, which were privately declared inside the enumeration definition and instantiated using the private constructor definition.

Other than working with the explicit list of enumeration values only (instead of an arbitrary number of objects of a class type), and needing to make our constructors private (or package-private), we essentially are writing a class with special restraints. We can even add a main method to an enumeration and use that as our entire Java program, just like any class's main method.