Chapter 4: Enumeration

Enumeration

Another useful tool in Java is *enumeration*, which is a way to define both a data type and all the possible values it can hold. Its name is often shortened to *enum*, which is the keyword used to define it.

An enumeration's values are all constants, and variables of that type are limited to those values. Enums are useful when you want to have data types with a few fixed values. Common uses include types for the days of the week and the months of the year.

I think an example will help clarify this. Here's a simple Java enumeration that defines the days of the week in order:

```
public enum DayOfWeek {
  SUNDAY,
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY;
}
```

This enum defines a new data type named DayOfWeek, which I can use in a program. If I create a DayOfWeek object, it can contain any of the seven values I declared for it. If you try to assign any value to a Day variable besides one of the seven values listed, the compiler will give you an error.

A declaration and assignment might look like this:

```
DayOfWeek day = DayOfWeek.MONDAY;
```

The values in an enum are similar to static constants like Math.PI. That's why programmers traditionally name them in all capital letters. You'd reference them like static constants, too, using their class name followed by the constant name, as in DayOfWeek.MONDAY.

Here's an example of a program using an enum like DayOfWeek to help you see how you might use enums in your programs. I'm going to write a switch statement similar to the one in the last example I gave you, but this time the switch cases will use an enum's values. Here's one way that might look:

```
/**
 * EnumTest1 runs a simple program as an example of using an enum in a switch
statement.
 *
 * @author Merrill Hall
 * @version 1.0
 */
public class EnumTest1 {
    public static void main(String[] args) {
        DayOfWeek day = DayOfWeek.MONDAY;
        howsYourDay(day);
        day = DayOfWeek.WEDNESDAY;
        howsYourDay(day);
        day = DayOfWeek.FRIDAY;
        howsYourDay(day);
        day = DayOfWeek.SATURDAY;
        howsYourDay(day);
        day = DayOfWeek.SUNDAY;
        howsYourDay(day);
    }

    public static void howsYourDay(DayOfWeek day) {
        switch (day) {
            case SUNDAY:
            case SATURDAY:
            System.out.println("It's " + day + "!  It's the weekend! Hooray!");
            break;
            case MONDAY:
            System.out.println("It's " + day + ". I'm sorry!");
            break;
            case FRIDAY:
            System.out.println("It's " + day + "! Only one day to go!");
            break;
            case WEDNESDAY:
            System.out.println("It's " + day + "! You're halfway there!");
            break;
            case TUESDAY:
            case THURSDAY:
            System.out.println("It's " + day + ". Nothing special about today.");
            break;
        }
        System.out.println();
```

```
        }
}
```

I want to point out two things about this little program.

- As you can see, enums are structures that you can use in a switch structure as the case values you want to check for.

- Each output statement uses the enum variable for output. Java will convert an enum variable into a string with its value when concatenated to a string or used in output (or both, as in this example). As we discussed back in Lesson 5, when you concatenate part of a program, you attach a second string to the end of a first string.

If you have any questions about the program, please post it in the Discussion Area.

At the risk of digging a little too deeply into their structure, let me mention just a few more things about enums. Java's enums also contain some built-in methods that are very useful. The first one is compareTo(). Since enum values are in a specified order, Java can compare them for you and tell you if one comes before or after another.

The compareTo() method will return a negative number if the first comes before the second, a positive number if the first comes after the second, and zero if they're the same value. For example,

```
DayOfWeek.TUESDAY.compareTo(DayOfWeek.FRIDAY)
```

returns a negative number, while

```
DayOfWeek.TUESDAY.compareTo(DayOfWeek.SUNDAY)
```

returns a positive number, and

```
DayOfWeek.TUESDAY.compareTo(DayOfWeek.TUESDAY)
```

returns zero. So this logic would tell you which of two days comes first:

```
DayOfWeek day1;
DayOfWeek day2;
 . . . // values are assigned somehow


if (day1.compareTo(day2) < 0)
  System.out.println(day1 + " comes before " + day2);
else if (day1.compareTo(day2) > 0)
  System.out.println(day1 + " comes after " + day2);
else
  System.out.println(day1 + " and " + day2 + " are the same");
```

Another useful method for an enum is toString(), which will return a string containing the value. So after this declaration:

```
DayOfWeek day = DayOfWeek.FRIDAY;
```

the call

```
day.toString()
```

would return the value "FRIDAY". So if I need to display the value of an enum variable that came from another source, and which I didn't set, I can use toString() to do that.

Java's enumerations also let you add your own data or methods. Let me show you an example that uses an enum of the planets in our solar system. In addition to its enum properties, each planet's value will have its relative gravity when compared to Earth so we can calculate relative weights. Here's the enum, and then I'll explain it and show you how to use it.

```
/**
 * Enumeration class Planet - lists the planets of the solar system and
 * their weight multiples relative to Earth.
 *
 * @author Merrill Hall
 * @version 1.0
 */
public enum Planet {
    MERCURY (0.378),
    VENUS   (0.907),
    EARTH   (1.0),
    MARS    (0.377),
    JUPITER (2.364),
    SATURN  (0.916),
    URANUS  (0.889),
    NEPTUNE (1.125),
    PLUTO   (0.067);

    private double relativeWeight;   // multiple to obtain weight

    private Planet(double relativeWeight) {
        this.relativeWeight = relativeWeight;
    }

    private double getRelativeWeight() {
        return relativeWeight;
    }

    public double getWeight(double weight) {
        return weight * relativeWeight;
    }
}
```

Yes, I still consider Pluto a planet! I grew up in a solar system with nine planets, and I like it that way!

But Pluto aside, this enum has several new features. To start with, instead of just names for the enumerated values, there are numeric values in parentheses. There's also an instance variable declared, a constructor, and two other methods. Let's discuss about each of these.

In this enumeration, the planet names are the enum values that the Planet type permits.

The instance variable, relativeWeight, is a numeric value attached to each enum value. It allows us to store information about that value along with it. The number represents the fraction of Earth's gravity that the given planet has, so we can calculate someone's weight on each of those planets.

The constructor, in this case and in all enumerations, is private instead of public like we've been using. That means that you can't create any Planet objects outside this class, and the Java compiler enforces that by making sure that all enum constructors are private. Only the list of enum values makes calls to the constructor, and that list also provides the value for the relativeWeight number for each planet name. So the line

```
    MERCURY (0.378),
```

creates an enum value named MERCURY with a relativeWeight value of 0.378. That means someone who weighs 100 pounds on Earth would weigh 37.8 pounds on Mercury.

Finally the two methods let us see each planet's weight factor relative to Earth's and calculate the weight on that planet given a weight from Earth.

Now how can we use all that additional information? I thought you'd never ask. Or maybe you wish I'd wrap this up! I will, right after I show you how to use this enum in a little program. Here it is:

```
import java.util.Scanner;
/**
 * PlanetaryWeights takes in a user's weight and returns
 * what that user would weigh on the different planets in the
 * solar system. It uses the enumeration Planet.
 *
 * @author Merrill Hall
 * @version 1.0
 */
public class PlanetaryWeights {
    public static void main(String[] args) {
        double earthWeight = 0;
        Scanner in = new Scanner(System.in);

        System.out.print("Enter your weight on Earth: ");
        if (in.hasNextDouble()) {
            earthWeight = in.nextDouble();
            for (Planet p : Planet.values())
                System.out.println("Your weight on " + p + " is " +
                                   p.getWeight(earthWeight));
        }
        else {
            System.out.println(in.next() + " is not a valid number.");
        }
    }
}
```

This program is simple. It declares a variable named earthWeight to capture the weight a user enters, and it sets up a Scanner variable called in to read the user's input for us.

After prompting the user, the program checks for numeric input. If the input isn't a number, then the program displays an error and quits.

If the input is a number, then the program reads it and stores it in the variable earthWeight.

The next statement is the only new type in this program. It's a loop format in Java that deals with a list or other collection, and it guarantees to process each item in the collection exactly once. If the collection has an order, as lists do, then the program will process the items from the collection in that order. Here's the statement:

```
            for (Planet p : Planet.values())
```

This is a *for-each loop* because it repeats once for every item in the collection. You could read this line as saying, "For each planet item in the list, . . ." where the dots represent the action you want to perform.

The "Planet p" before the colon tells Java two things: which types of items are in the list (Planet), and the name I want to use for each item as I process it (p).

The part of the statement after the colon is the source of the list whose items I want to process. In this case, it's a call to a method named values() that Java adds to every enumeration. That method produces a list of all the enum values. So in this case, it returns a list of the nine planet names, and the for-each loop gives them to us one at a time in the variable p so we can act on each one.

The action we're taking is simple—it's just an output statement with some text, the value of the Planet item we're dealing with, and a call to that planet's getWeight method to convert the weight from Earth dimensions to that planet's weight. If I run this program using my current weight of 190 pounds, here's the output I'll see:

```
Enter your weight on Earth: 190
Your weight on MERCURY is 71.82000000000001
Your weight on VENUS is 172.33
Your weight on EARTH is 190.0
Your weight on MARS is 71.63
Your weight on JUPITER is 449.15999999999997
Your weight on SATURN is 174.04000000000002
Your weight on URANUS is 168.91
Your weight on NEPTUNE is 213.75
Your weight on PLUTO is 12.73
```

A few of those weights are poorly formatted due to the way computers store floating-point numbers. In the next lesson I'll show you a way in Java to produce exactly the format we want for numbers.