Lecture - Enumerations

Enumerated types counting things.

Starting with version 5.0, Java permits enumerated types

An enumerated type is a type in which all the values are given in a (typically short) list

The definition of an enumerated type is normally placed outside of all methods in the same place that named
constants are defined:

```
enum TypeName {VALUE_1, VALUE_2, ..., VALUE_N};
```

Note that a value of an enumerated type is a kind of named constant and so, by convention, is spelled with all UPPERCASE_LETTERS, with underlines between them.

As with any other type, variables can be declared of an enumerated type. Given the following definition

```
enum WorkDay {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};
```

variables can be declared as

```
WorkDay meetingDay, availableDay;
```

The value of a variable of this type can be set to one of the values listed in the definition of the type, or the special value `null`:

```
meetingDay = WorkDay.THURSDAY;
availableDay = null;
```

Enumerated types usage

Although they may look like String values, values of an enumerated type are not String values

However, they can be used for tasks which could be done by String values and, in some cases, work better:

- Using a String variable allows the possibility of setting the variable to a nonsense value. Using an enumerated type variable constrains the possible values for that variable. An error message will result if an attempt is made to give an enumerated type variable a value that is not defined for its type
- Enumerated types are also more efficient. An enumerated type is stored as an integer. The computer can test if the enumerated type has a particular value in one "clock cycle", whereas testing if strings are equal takes many clock cycles; more for longer strings.

Two variables or constants of an enumerated type can be compared using the `equals` method or the == operator. However, the == operator has a nicer syntax:

```
if (meetingDay == availableDay)
    System.out.println("Meeting will be on schedule.");
if (meetingDay == WorkDay.THURSDAY)
    System.out.println("Long weekend!);
```

Enumerated Types in switch

Enumerated types can be used to control a switch statement.

The switch control expression uses a variable of an enumerated type.

Case labels are the *unqualified* values of the same enumerated type (i.e., they don't mention the type name).

```
import java.util.Scanner;

public class Main {
    enum Flavour {VANILLA, CHOCOLATE, strawberry};   put the constraint

    public static void main (String[] args) {
        Flavour favourite = null;
        Scanner keyboard = new Scanner(System.in);

        System.out.println("What is your favourite flavour?");
        for (Flavour f : Flavour.values())   for each loop
            System.out.print(" " + f);
        System.out.println();
```

```
    System.out.print(" " + f);
String vanName = Flavour.VANILLA.name();
```

```
        String answer = keyboard.next();
        favourite = Flavour.valueOf(answer);

        switch (favourite) {
            case VANILLA:
                System.out.println("Classic");
                break;
            case CHOCOLATE:
                System.out.println("Rich");
            default:
                System.out.println("I bet you said strawberry.");
                break;
        }
    }
}
```

This example also uses three new features of enumerated types.

- The static method `Flavour.values()` returns an array of type `Flavour` containing each value of the enumeration.
- Enumeration values convert to String in the way you would expect: the string value is a string containing the same name as is used in code. For example `Flavour.CHOCOLATE.toString()` is "CHOCOLATE". (Note that `toString()` is called implicitly when a value has to be converted to a `String`, such as when it is being added to a `String`.)
- The reverse -- converting from a string to the enumeration value -- is done by `Flavour.valueOf()`. The input is a string, which must be ***exactly*** the name of the enumeration value. The case must match, and it must not have any spaces.

```
Flavour.valueOf(Flavour.CHOCOLATE.toString()) == Flavour.CHOCOLATE
```

EXERCISE: (challenging) Modify the code above to report an error if the string entered is not one of the values of `Flavour`.

Enumeration methods

The following are some methods that every enumerated type has automatically.

`protected Enum(String name, int ordinal)`

This is the only constructor. However enumerations are like primitive types, and can be just assigned from literals, like Flavour.VANILLA without needing a `new` and a constructor.

`boolean equals(Object other)`

Returns true if the specified object is equal to this enum constant.

`String toString()`

`String name()`

Returns the name of this enum constant, as contained in the declaration. The difference between these two is that `toString()` can be *overridden* (a process which we will cover in a later lecture) but `name` cannot.

`int ordinal()`

Returns the ordinal of this enumeration constant (its position in its enum declaration, where the initial constant is assigned an ordinal of zero).

`int compareTo(`*`EnumeratedType`*` o)`

Compares this enum with the specified object. Returns > 0 if this is later in the list than `o`, or < 0 if this is earlier or 0 if they are equal..

`public `*`EnumeratedType`*` [] values ()`

Returns an array whose elements are the values of the enumerated type in the order in which they are listed in the definition of the enumerated type.

`static `*`EnumeratedType`*` valueOf(String name)`

Returns the enum constant of the specified enum type with the specified name.