# Enumerations and Nested Classes

**Enumerations** An **enumeration** (enum) is a data type that contains a fixed set of constants. Enums are good to use when you already know all possibilities of the values or instances of the class. They can be declared on their own or within another class

With numeric or String constants, you can pass an invalid value and not find out until runtime. With enums, it is impossible to create an invalid enum value without introducing a compiler error.

Enumerations show up whenever you have a set of items whose types are known at compile time. Common examples include the following:

* Compass directions
* Months of the year
* Cards in a deck

To create an enum, declare a type with the **enum** keyword, an identifier for the type, and a list of values called **enum constants**. The enum constants are the only allowed values for the type.

**public enum Period {**

**PRELIM, MIDTERM, PREFINAL, FINAL;**

**}**

After creating an enumerated data type, you can declare variables of that type just like the following: **Period p;**

You can assign any of the enum constants to the variable, such as the following statement: **p = Period.MIDTERM;**

An enumeration type such as *Period* is a class. Its enum constants act like objects instantiated from the class, including having access to the methods of the class. These built-in methods are non-static.

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Example if p = Period.MIDTERM;** |
| toString() | Returns the name of the calling constant object | **p.toString()** is  "MIDTERM" |

|  |  |  |
| --- | --- | --- |
| ordinal() | Returns an integer that represents the constant’s position in the list of constants; the first position is 0 | **p.ordinal()** is 1 |
| equals() | Returns true if its argument is equal to the calling object’s value | **p.equals(Period.PR ELIM)**is false |
| compareTo() | Returns a negative integer if the calling object’s ordinal value is less than that of the argument, 0 if they are the same, and a positive integer if the calling object’s ordinal value is greater than that of the  argument | **p.compareTo(Period**  **.FINAL)**is negative  **p.compareTo(Period**  **.PRELIM)**is positive  **p.compareTo(Period**  **.MIDTERM)**is 0 |

You also can use comparison operators with enumeration constants instead of using the compareTo() method to return a number. Several static methods can also be used with enumerations. These are used with the type and not with the individual constants. Below are two (2) useful static methods to use with enumerations.

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Example with Period Enumeration** |
| valueOf() | Accepts a string parameter and returns an enumeration  constant | **Period.valueOf("PRELIM")** returns the PRELIM enum constant |
| values() | Returns an array of the enumerated constants | **Period.values()** returns an  array with 4 elements that contain the enum constants |

You can declare an enumerated type in its own file, where the filename matches the type name and has a .java extension. You can also declare an enumerated type within a class but not within a method. In the following example, a *Period* enumeration is declared, while a *Period* variable is declared in the main method. Then, an enhanced for loop declares a local *Period* variable named p that takes on the value of each element in the Period.values() array in turn so it can be displayed.

You can also use enumerations to control a switch structure. The following example contains a class that declares an enumeration for pizza sizes and then uses a switch structure to display the prices.

**import java.util.Scanner; public class EnumDemo {**

**enum Period {PRELIM, MIDTERM, PREFINAL, FINAL};**

**public static void main(String[] args) { Period pd;**

**String input; int position; int comparison;**

**Scanner sc = new Scanner(System.in); System.out.println("The grading periods are:"); for(Period p : Period.values()) System.out.print(p + " "); System.out.print("\nSelect a grading period: "); input = sc.nextLine().toUpperCase();**

**pd = Period.valueOf(input); System.out.println("You entered " + pd); position = pd.ordinal(); System.out.println(pd + " is in position " +**

**position);**

**System.out.println("So its period number is " + (position + 1));**

**}**

**}**

**import java.util.Scanner; public class EnumDemo2 {**

**enum Size {REGULAR, LARGE, PARTY};**

**public static void main(String[] args) { Size s = Size.LARGE; System.out.print("Price of pizza is "); switch(s) {**

**case REGULAR: System.out.println("381.00"); break;**

**case LARGE: System.out.println("602.00"); break;**

**case PARTY: System.out.println("799.00"); break;**

**}**

**}**

**}**

The user is prompted to select a period, which is converted to its uppercase equivalent. The valueOf() method is used to convert the user’s input to an enumeration value. The program also retrieves the position of the selected period in the enumeration list.

Below is a sample output of the above code.

**The grading periods are: PRELIM MIDTERM PREFINAL FINAL**

**Select a grading period: Midterm You entered MIDTERM**

**MIDTERM is in position 1 So its period number is 2**

In the above program, s is assigned the LARGE constant, so the output is:

# Price of pizza is 602.00

Using enums makes the values type-safe; **type-safe** is used to describe a data type for that allows only appropriate behaviors. As with other classes, you can also add methods and other fields to an enum type.

**Nested Classes** In Java, you can create a class within another class and store them together; such classes are called **nested classes**. The containing class is the top-level class. There are four (4) types of nested classes:

* **static member class** – has access to all static methods of the top- level class
* **Non-static member class (inner class)** – requires an instance; it has access to all data and methods of the top-level class.
* **Local class** – defined within a method body
* **Anonymous class** – a special case of a local class that has not identifier

The most common reason for nesting a class inside another is because the inner class is used only by the top-level class. Being able to package the classes together makes their connection easier to understand and their code easier to maintain.

The sample class below describes houses that are being sold by a real estate company. The class might contain separate fields for the listing number, price, address, and area. Although the listing number and price go with the real estate listing, you might also decide that the address and area really go with the house. You might create an inner class like the one in the sample program below.

**public class RealEstateListing { private int listingNumber; private double price;**

**private HouseData hData;**

**public RealEstateListing(int num, double price, String address, int sqFt) {**

**listingNumber = num; this.price = price;**

**hData = new HouseData(address, sqFt);**

**}**

**public void display() {**

**System.out.println("Listing number #" + listingNumber + " Selling for Php" + price);**

**System.out.println("Address: " + hData.streetAddress); System.out.println(hData.squareFeet + " square feet");**

**}**

**private class HouseData { private String streetAddress; private int squareFeet;**

**public HouseData(String address, int sqFt) { streetAddress = address;**

**squareFeet = sqFt;**

**}**

**}**

}

In the sample program, the inner *HouseData* class is a private class. You don’t have to make an inner class private, but doing so keeps its members hidden from outside classes. If you wanted a class’s members to be accessible, you would not make it an inner class. An inner class can access its top-level class’s fields and methods, even if they are private, and an outer class can access its inner class’s members.

You usually will not want to create inner classes. For example, if you made the *HouseData* class a regular class (as opposed to an inner class) and stored it in its own file, you could use it with composition in other classes. As it stands, it is usable only in the class in which it now resides. You probably will not create nested classes frequently, but you will see them implemented in some built-in Java classes.