Programming in Java

Lecture 16: Enumerations and Autoboxing

Mahesh Kumar
Assistant Professor (Adhoc)

Department of Computer Science
Acharya Narendra Dev College
University of Delhi

Course webpage
[http://www.mkbhandari.com/mkwiki]

Outline

1 Enumerations

2 Type Wrappers

- 2.1 Autoboxing
- 2.2 Unboxing

Enumerations

- An Enumeration is a list of named constants.
- Java enumerations is similar to enumerations in other languages with some differences
- In Java, an enumeration defines a class type. By making enumerations into classes, the capabilities of the enumeration are greatly expanded.
- In Java, an enumeration can have constructors, methods, and instance variables.
- An enumeration is created using the ***enum*** keyword. For example, here is a simple enumeration that lists various apple varieties:

```
// An enumeration of apple varieties.
enum Apple {
        Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

Enumerations

```
// An enumeration of apple varieties.
enum Apple {
        Jonathan, GoldenDel, RedDel,
                        Winesap, Cortland
}
```

1 *The identifiers Jonathan, GoldenDel, and so on, are called enumeration constants.*

2 *Each is implicitly declared as a public, static final member of Apple.*

3 *Once you have defined an enumeration, you can create a variable of that type. However, even though enumerations define a class type, you do not instantiate an **enum** using **new.***

```
Apple ap;       //same as in primitive types.
```

4 *Because ap is of type Apple, the only values that it can be assigned (or can contain) are those defined by the enumeration.*

```
ap = Apple.RedDel;
```

Enumerations

```
// An enumeration of apple varieties.

enum Apple {
        Jonathan, GoldenDel, RedDel, Winesap, Cortland;
}

class EnumDemo {
        public static void main(String args[ ]) {
                Apple ap;
                ap = Apple.RedDel;
                // Output an enum value.
                System.out.println("Value of ap: " + ap);
                System.out.println( );
                ap = Apple.GoldenDel;
                // Compare two enum values.
                if(ap == Apple.GoldenDel){
                        System.out.println("ap contains GoldenDel.\n");
                }
```

```
// Use an enum to control a switch statement.
switch(ap) {
        case Jonathan:
                System.out.println("Jonathan is red.");
                break;
        case GoldenDel:
                System.out.println("Golden Delicious is yellow.");
                break;
        case RedDel:
                System.out.println("Red Delicious is red.");
                break;
        case Winesap:
                System.out.println("Winesap is red.");
                break;
        case Cortland:
                System.out.println("Cortland is red.");
                break;
        }
        }
}
```

Enumerations

```
    // Use an enum to control a switch statement.
    switch(ap) {
        case Jonathan:
            System.out.println("Jonathan is red.");
            break;
        case GoldenDel:
            System.out.println("Golden Delicious is yellow.");
            break;
        case RedDel:
            System.out.println("Red Delicious is red.");
            break;
        case Winesap:
            System.out.println("Winesap is red.");
            break;
        case Cortland:
            System.out.println("Cortland is red.");
            break;
        }
    }
}
```

The output from the program is shown here:

```
Value of ap: RedDel
ap contains GoldenDel.
Golden Delicious is yellow.
```

The *values( )* and *valueOf( )* Methods

- All enumerations automatically contain two predefined methods:

  1. public static enum-type [ ] values( )
     - *The values( ) method returns an array that contains a list of the enumeration constants.*

  2. public static enum-type valueOf(String str)
     - *The valueOf( ) method returns the enumeration constant whose value corresponds to the string passed in str.*

- In both cases, enum-type is the type of the enumeration.

The *values( )* and *valueOf( )* Methods

```
// Use the built-in enumeration methods.
// An enumeration of apple varieties.
enum Apple {
        Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
class EnumDemo2 {
        public static void main(String args[ ]){
                Apple ap;
                System.out.println("Here are all Apple constants:");

                // use values( )
                Apple allapples[ ] = Apple.values( );

                for(Apple a : allapples)
                        System.out.println(a);

                System.out.println( );

                // use valueOf( )
                ap = Apple.valueOf("Winesap");
                System.out.println("ap contains " + ap);
        }
}
```

The output from the program is shown here:

```
Here are all Apple constants:
Jonathan
GoldenDel
RedDel
Winesap
Cortland

ap contains Winesap
```

Java Enumerations Are Class Types

- Java enumeration is a class type.
  - *Although you can't instantiate an* ***enum*** *using* ***new****, it otherwise has much the same capabilities as other classes.*

- Enumeration can have constructors, instance variables and methods:
  - *Each enumeration constant is an object of its enumeration type*
  - *The constructor is called when each enumeration constant is created*
  - *Each enumeration constant has its own copy of any instance variables defined by the enumeration*

Java Enumerations Are Class Types

```
// Use an enum constructor, instance variable, and method.
enum Apple {
        Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8); // Arguments for Constructors
        private int price;  // Price of each apple
        Apple(int p) {      // Constructor
                price = p;
        }
        int getPrice( ) {   // Method
                return price;
        }
}
class EnumDemo3 {
        public static void main(String args[ ]) {
        Apple ap;
        // Display price of Winesap.
        System.out.println("Winesap costs " +
                        Apple.Winesap.getPrice( ) +" cents. \n");
        // Display all apples and prices.
        System.out.println("All apple prices:");
        for(Apple a : Apple.values( ))
                System.out.println(a + " costs " + a.getPrice( ) +" cents.");
        }
}
```

The output is shown here:

```
Winesap costs 15 cents.

All apple prices:
Jonathan costs 10 cents.
GoldenDel costs 9 cents.
RedDel costs 12 cents.
Winesap costs 15 cents.
Cortland costs 8 cents.
```

- Enumerations Inherit Enum

  // Self Study Page No. 269-272

Type Wrappers

- Java uses primitive types (also called simple types), such as ***int*** or ***double***, to hold the basic data types supported by the language.

- Primitive types, rather than objects, are used for these quantities for the sake of performance.

- Using objects for these values would add an unacceptable overhead to even the simplest of calculations.

- Thus, the primitive types are not part of the object hierarchy, and they do not inherit ***Object***.

- Despite the performance benefit offered by the primitive types, there are times when you will need an object representation
  - *You can't pass a primitive type by reference to a method*
  - *Many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types*

Type Wrappers

- Java provides **type wrappers**
  - *classes that encapsulate a primitive type within an object*

- The type wrappers are:
  - *Character*
  - *Boolean*
  - *Double, Float, Long, Integer, Short, Byte*

- These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

Type Wrappers

1 ***Character***

- ***Character*** is a wrapper around a ***char***. The constructor for ***Character*** is:

  Character(char ch)

  ch specifies the character that will be wrapped by the ***Character*** object being created.

- To obtain the ***char*** value contained in a ***Character*** object, call charValue( ), shown here:

  ```
  char charValue( )    // It returns the encapsulated character.
  ```

Type Wrappers

2 ***Boolean***

- ***Boolean*** is a wrapper around boolean values. It defines these constructors:

  Boolean(boolean boolValue) // ***boolValue*** must be either ***true*** or ***false***.

  Boolean(String boolString) // if ***boolString*** contains the string **"true"** (in uppercase or lowercase), then the new ***Boolean*** object will be true. Otherwise, it will be false.

- To obtain a ***boolean*** value from a ***Boolean*** object, use booleanValue( ), shown here:

  boolean booleanValue( ) // It returns the **boolean** equivalent of the invoking object.

Type Wrappers

3 *The Numeric Type Wrappers*

- By far, the most commonly used type wrappers are those that represent numeric values. These are ***Byte, Short, Integer, Long, Float,*** and ***Double***.
- All of the numeric type wrappers inherit the abstract class ***Number***.
- ***Number*** declares methods that return the value of an object in each of the different number formats. These methods are shown here:

```
byte byteValue( )
double doubleValue( )     // doubleValue( ) returns the value of an object as a double.
float floatValue( )       // floatValue( ) returns the value as a float, and so on.
int intValue( )
long longValue( )
short shortValue( )
```

- These methods are implemented by each of the numeric type wrappers.

Type Wrappers

- The following program demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value.

```
// Demonstrate a type wrapper.

class Wrap {
        public static void main(String args[ ]) {

                // The process of encapsulating a value within an object is called boxing.
                Integer iOb = new Integer(100)        // Wraps the integer value 100 inside an Integer object called iOb.


                //The process of extracting a value from a type wrapper is called unboxing.
                int i = iOb.intValue( );     // Obtains the value by calling intValue( ) and stores the result in i.


                System.out.println(i + "  " + iOb);      // displays 100 100
        }
}
```

Auto (boxing/unboxing)

- ## Autoboxing
  - The process by which a primitive type is automatically encapsulated into its equivalent type wrapper whenever an object of that type is needed.
  - There is no need to explicitly construct an object.

- ## Auto-unboxing
  - The process by which the value of a boxed object is automatically extracted from a type wrapper when its value is needed
  - There is no need to call a method such as intValue( ) or doubleValue( ).

Autoboxing

- With autoboxing, it is no longer necessary to manually construct an object in order to wrap a primitive type
- You need only assign that value to a type-wrapper reference
- Java automatically constructs the object for you:
  - ***Integer iOb = 100; // autobox an int***   **100**
- Notice that the object is not explicitly created through the use of new. Java handles this for you, automatically

```
// Demonstrate autoboxing/unboxing.

class AutoBox {
        public static void main(String args[ ]) {
                Integer iOb = 100;        // autobox an int
                int i = iOb;              // auto-unbox
                System.out.println(i + " " + iOb);      // displays 100 100
        }
}
```

Auto-unboxing

- To unbox an object, simply assign that object reference to a primitive-type variable
  - ***int i = iOb;***   ***// auto-unbox***

- Java handles the details for you

// Self Study following topics from Page No. 275-279

- Autoboxing and Methods
- Autoboxing/Unboxing Occurs in Expressions
- Autoboxing/Unboxing Boolean and Character Values
- Autoboxing/Unboxing Helps Prevent Errors

References

Reference for this topic

- [Book: Java: The Complete Reference, Ninth Edition: Herbert Schildt]
  https://www.amazon.in/Java-Complete-Reference-Herbert-Schildt/dp/0071808558

- [Web: GeeksforGeeks]
  https://www.geeksforgeeks.org/java/

- [Web: Java T Point tutorial]
  https://www.javatpoint.com/java-tutorial