Module -5

Chapter -2: Enumerations, Autoboxing, and Annotations

Enumerations

- Enumerations in Java provide a structured way to define a new data type with named constants representing legal values.
- They offer a more robust alternative to using final variables for defining constant values. Enumerations are commonly used to represent sets of related items, such as error codes or states of a device.
- In Java, enumerations are implemented as classes, allowing for constructors, methods, and instance variables, which greatly enhances their capabilities and flexibility.
- They are extensively used throughout the Java API library due to their power and versatility.

Enumeration Fundamentals

1. **Definition**:
   - Enumerations are created using the `enum` keyword.
   - Constants within the enumeration are called enumeration constants.
2. **Constants Declaration**:
   - Enumeration constants are implicitly declared as public, static, final members of the enumeration type.
   - Each constant is of the type of the enumeration in which it is declared.
3. **Instantiation**:

- Enumerations define a class type, but they are not instantiated using the `new` keyword.
- Enumeration variables are declared and used similarly to primitive types.

4. **Assignment and Comparison**:
   - Enumeration variables can only hold values defined by the enumeration.
   - Constants can be assigned to enumeration variables using the dot notation (`EnumType.Constant`).
   - Constants can be compared for equality using the `==` relational operator.
5. **Switch Statements**:
   - Enumeration values can be used to control switch statements.
   - All case statements within the switch must use constants from the same enum as the switch expression.
   - Constants in case statements are referenced without qualification by their enumeration type name.
6. **Displaying Enumeration Constants**:
   - Enumeration constants are displayed by outputting their names.
   - Enumeration constants are referenced using the dot notation (`EnumType.Constant`)

The following program puts together all of the pieces and demonstrates the **Apple** enumeration:

```
// 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;
        }
    }
}
```

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: **values( )** and **valueOf( )**.
- Their general forms are shown here:

  public static *enum-type* [ ] values( )
  public static *enum-type* valueOf(String *str*)

- The **values( )** method returns an array that contains a list of the enumeration constants.
- 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.
- For example, in the case of the **Apple** enumeration shown earlier, the return type of **Apple.valueOf("Winesap")** is **Winesap**.

The following program demonstrates 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:
Jonat
han
Golde
nDel
RedD
el
Wine
sap
Cortl
and

ap contains Winesap
```

Notice that this program uses a for-each style **for** loop to cycle through the array of constants obtained by calling **values( )**. For the sake of illustration, the variable **allapples** was created and assigned a reference to the enumeration array. However, this step is not necessary because the **for** could have been written as shown here, eliminating the need for the **allapples** variable:

```
for(Apple    a    :    Apple.values())
    System.out.println(a);
```

Now, notice how the value corresponding to the name **Winesap** was obtained by calling valueOf( ).

```
ap = Apple.valueOf("Winesap");
```

Java Enumerations Are Class Types

1. **Enum as Class Type**:
    - Java enumerations are treated as class types.
    - They have similar capabilities as other classes, allowing constructors, instance variables, methods, and interface implementations.
2. **Enumeration Constants**:
    - Each enumeration constant is an object of its enumeration type.

- Constructors can be defined for enums, and they are called when each enumeration constant is created.
- Instance variables defined within the enum are associated with each enumeration constant separately.

3. **Example with Apple Enum**:
   - An example is provided with an `Apple` enum representing different varieties of apples.
   - Each constant has an associated price stored in an instance variable.
   - A constructor `Apple(int p)` initializes the price for each apple variety.
   - A method `getPrice()` returns the price of the apple variety.
4. **Usage Example**:
   - In the `main()` method, the prices of different apple varieties are displayed.
   - The constructor is called for each enumeration constant to initialize the prices.
   - Instance methods can be called on enumeration constants to retrieve associated data.
5. **Overloaded Constructors**:
   - Enumerations can have two or more overloaded constructors, just like other classes.
   - An example is provided with a default constructor initializing the price to -1 when no price data is available.

```
// Use an enum constructor. enum Apple {
    Jonathan(10), GoldenDel(9), RedDel, Winesap(15), Cortland(8); private int price; // price of each

    apple

    // Constructor
    Apple(int p) { price = p; }

    // Overloaded constructor Apple() { price
    = -1; }

    int getPrice() { return price; }
}
```

Enumerations Inherit Enum

1. **Inheritance**:
   - All Java enumerations automatically inherit from the `java.lang.Enum` class.
   - While you can't inherit a superclass when declaring an enum, `java.lang.Enum` is implicitly inherited by all enums.
2. `java.lang.Enum` **Methods**:
   - `ordinal()` **Method**:
     - Returns the ordinal value of the invoking enumeration constant.
     - Ordinal values start at zero and increase sequentially.
     - Example: `Apple.Jonathan.ordinal()` would return 0.
   - `compareTo()` **Method**:
     - Compares the ordinal value of the invoking constant with another constant of the same enumeration.
     - Returns a negative value if the invoking constant's ordinal value is less than the other constant's, zero if they're equal, and a positive value if it's greater.
     - Example: `Apple.Jonathan.compareTo(Apple.RedDel)` would return a negative value.
   - `equals()` **Method**:
     - Overrides the `equals()` method defined by `Object`.
     - Compares an enumeration constant with any other object.
     - Returns true only if both objects refer to the same constant within the same enumeration.
     - Example: `Apple.Jonathan.equals(Apple.Jonathan)` would return true.

3. **Comparing Enumeration Constants**:
   - Enumeration constants can be compared for equality using the `==` operator.
   - The `equals()` method can also be used to compare constants for equality, ensuring they belong to the same enumeration.
4. **Example Program**:
   - Demonstrates the usage of `ordinal()`, `compareTo()`, and `equals()` methods with enumeration constants.

```
// Demonstrate ordinal(), compareTo(), and equals().

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

class EnumDemo4 {
    public static void main(String[] args)
    {
        Apple ap, ap2, ap3;

        // Obtain all ordinal values using ordinal(). System.out.println("Here are all apple constants" +
                                        " and their ordinal values: "); for(Apple a : Apple.values())
            System.out.println(a + " " + a.ordinal());

        ap    =    Apple.RedDel;    ap2    =
        Apple.GoldenDel;        ap3        =
        Apple.RedDel;

        System.out.println();

        // Demonstrate compareTo() and equals() if(ap.compareTo(ap2) < 0)
            System.out.println(ap + " comes before " + ap2);

        if(ap.compareTo(ap2) > 0)
            System.out.println(ap2 + " comes before " + ap);

        if(ap.compareTo(ap3)  ==  0)  System.out.println(ap  +  "  equals  "  +
            ap3);

        System.out.println(); if(ap.equals(ap2))
            System.out.println("Error!");

        if(ap.equals(ap3))
            System.out.println(ap + " equals " + ap3);
            if(ap == ap3)
                System.out.println(ap + " == " + ap3);

        }
    }
```

The output from the program is shown here:

```
Here are all apple constants and their ordinal values: Jonathan 0
GoldenDel 1
RedDel 2
Winesap 3
Cortland 4

GoldenDel  comes  before  RedDel  RedDel
equals RedDel

RedDel     equals     RedDel
RedDel == RedDel
```

Another Enumeration Example

- An automated “decision maker” program was created. In that version, variables called **NO**, **YES**, **MAYBE**, **LATER**, **SOON**, and **NEVER** were declared within an interface and used to represent the possible answers. While there is nothing technically wrong with that approach,
- the enumeration is a better choice. Here is an improved version of that program that uses an **enum** called **Answers** to define the answers.

```
// An improved version of the "Decision Maker"
// program from Chapter 9. This version uses an
// enum, rather than interface variables, to
//     represent     the     answers.     import

java.util.Random;

// An enumeration of the possible answers. enum Answers {
    NO, YES, MAYBE, LATER, SOON, NEVER
}

class Question {
    Random rand = new Random(); Answers ask() {
    int prob = (int) (100 * rand.nextDouble());

        if (prob < 15)
            return  Answers.MAYBE;  //  15%  else  if
        (prob < 30)
            return Answers.NO;                //    15%
        else if (prob < 60)
            return Answers.YES;               // 30%
        else if (prob < 75)
            return  Answers.LATER;  //  15%  else  if
        (prob < 98)
            return Answers.SOON;   // 13% else
            return Answers.NEVER; // 2%
    }
}

class AskMe {
    static void answer(Answers result) { switch(result) {
            case NO: System.out.println("No"); break;
            case YES: System.out.println("Yes"); break;
            case    MAYBE:    System.out.println("Maybe");
                break;
            case LATER: System.out.println("Later"); break;
            case SOON: System.out.println("Soon"); break;
            case NEVER: System.out.println("Never"); break;
        }
    }

    public  static  void  main(String[]  args)  {  Question  q  =  new
        Question(); answer(q.ask());
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
    }
}
```

Type Wrappers

- Java provides type wrapper classes that encapsulate primitive types within objects.
- Type wrapper classes include `Double`, `Float`, `Long`, `Integer`, `Short`, `Byte`, `Character`, and `Boolean`.
- These classes offer methods that allow integration of primitive types into Java's object hierarchy.
- Java's autoboxing feature automatically converts primitive types to their corresponding wrapper classes when necessary, and vice versa.
- This simplifies the process of working with both primitive types and objects, as conversions are handled implicitly by the compiler.

Overall, type wrappers in Java allow primitive types to be used in situations where objects are required, providing a bridge between the world of primitive types and the object-oriented nature of Java. Autoboxing further simplifies the interaction between primitive types and objects by handling conversions automatically.

Character

- **Character** is a wrapper around a **char**. The constructor for **Character** is Character(char *ch*)
- Here, *ch* specifies the character that will be wrapped by the **Character** object being created.
- However, beginning with JDK 9, the **Character** constructor was deprecated, and beginning with JDK 16, it has been deprecated for removal. Today, it is strongly recommended that you use the static method **valueOf( )** to obtain a **Character** object.
- It is shown here:
    - static Character valueOf(char *ch*)
- It returns a **Character** object that wraps *ch*.
- To obtain the **char** value contained in a **Character** object, call **charValue( )**, shown here: char charValue( )
- It returns the encapsulated character.

Boolean

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

Boolean(boolean *boolValue*)
Boolean(String *boolString*)

- In the first version, *boolValue* must be either **true** or **false**. In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.
- However, beginning with JDK 9, the **Boolean** constructors were deprecated, and beginning with JDK 16, they have been deprecated for removal. Today, it is strongly recommended that you use the static method **valueOf( )** to obtain a **Boolean** object. It has the two versions shown here:

static Boolean valueOf(boolean *boolValue*)
static Boolean valueOf(String *boolString*)

- Each returns a **Boolean** object that wraps the indicated value.
- To obtain a **boolean** value from a **Boolean** object, use **booleanValue( )**, shown here: boolean booleanValue( )
- It returns the **boolean** equivalent of the invoking object.

The Numeric Type Wrappers

- 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(  )
float floatValue( )
int     intValue(     )
long longValue( )
short shortValue( )
```

- For example, **doubleValue( )** returns the value of an object as a **double**, **floatValue( )** returns the value as a **float**, and so on.
- These methods are implemented by each of the numeric type wrappers.
- All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value.
- For example, here are the constructors defined for **Integer**:

```
Integer(int            num)
Integer(String str)
```

- If *str* does not contain a valid numeric value, then a **NumberFormatException** is thrown.
- Here are two of the forms supported by **Integer**:

```
static Integer valueOf(int val)
static Integer valueOf(String valStr) throws NumberFormatException
```

Here, *val* specifies an integer value and *valStr* specifies a string that represents a properly formatted numeric value in string form. Each returns an **Integer** object that wraps the specified value. Here is an example:

```
Integer iOb = Integer.valueOf(100);
```

After this statement executes, the value 100 is represented by an **Integer** instance. Thus, **iOb** wraps the value 100 within an object. In addition to the forms **valueOf( )** just shown, the integer wrappers, **Byte**, **Short**, **Integer**, and **Long**, also supply a form that lets you specify a radix.
All of the type wrappers override **toString( )**. It returns the human-readable form of the value contained within the wrapper. This allows you to output the value by passing a type wrapper object to **println( )**, for example, without having to convert it into its primitive type.

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)  {  Integer  iOb  =

      Integer.valueOf(100); int i = iOb.intValue();

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

This program wraps the integer value 100 inside an **Integer** object called **iOb**. The program then obtains this value by calling **intValue( )** and stores the result in **i**.
The process of encapsulating a value within an object is called *boxing*. Thus, in the program, this line boxes the value 100 into an **Integer**:

```
Integer iOb = Integer.valueOf(100);
```

The process of extracting a value from a type wrapper is called *unboxing*. For example, the program unboxes the value in **iOb** with this statement:

```
int i = iOb.intValue();
```

The same general procedure used by the preceding program to box and unbox values has been available for use since the original version of Java. However, today, Java provides a more streamlined approach, which is described next.

Autoboxing

- Modern versions of Java have included two important features: *autoboxing* and *auto-unboxing*.
- Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed.
- There is no need to explicitly construct an object. Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.
- There is no need to call a method such as **intValue( )** or **doubleValue( )**.
- Autoboxing and auto-unboxing greatly streamline the coding of several algorithms, removing the tedium of manually boxing and unboxing values.
- They also help prevent errors. Moreover, they are very important to generics, which operate only on objects. Finally, autoboxing makes working with the Collections Framework
- With autoboxing, it is not 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. For example, here is the modern way to construct an **Integer** object that has the value 100:

```
Integer iOb = 100; // autobox an int
```

Notice that the object is not explicitly boxed. Java handles this for you, automatically.

To unbox an object, simply assign that object reference to a primitive-type variable. For example, to unbox **iOb**, you can use this line:

```
int i = iOb; // auto-unbox
```

Java handles the details for you.

Here is the preceding program rewritten to use autoboxing/unboxing:

```
// 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
   }
}
```

Autoboxing and Methods

In addition to the simple case of assignments, autoboxing automatically occurs whenever a primitive type must be converted into an object; auto-unboxing takes place whenever an object must be converted into a primitive type. Thus, autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method. For example, consider this:

```
// Autoboxing/unboxing takes place with
// method parameters and return values.

class AutoBox2 {
   // Take an Integer parameter and return
   // an int value;
   static int m(Integer v) {
      return v ; // auto-unbox to int
   }

   public static void main(String[] args) {
      // Pass an int to m() and assign the return value
      // to an Integer.   Here, the argument 100 is autoboxed
      // into an Integer.   The return value is also autoboxed
      // into an Integer.
      Integer iOb = m(100);

      System.out.println(iOb);
   }
}
```

This program displays the following result:

```
100
```

Autoboxing/Unboxing Occurs in Expressions

In general, autoboxing and unboxing take place whenever a conversion into an object or from an object is required. This applies to expressions. Within an expression, a numeric object is automatically unboxed. The outcome of the expression is reboxed, if necessary. For example, consider the following program:

```
// Autoboxing/unboxing occurs inside expressions.

class AutoBox3 {
   public static void main(String[] args) {

      Integer iOb, iOb2; int i;

      iOb = 100;
      System.out.println("Original value of iOb: " + iOb);

      // The following automatically unboxes iOb,
      // performs the increment, and then reboxes
      // the result back into iOb.
      ++iOb;
      System.out.println("After ++iOb: " + iOb);

      // Here, iOb is unboxed, the expression is
      // evaluated, and the result is reboxed and
      // assigned to iOb2. iOb2 = iOb +
      (iOb / 3);
      System.out.println("iOb2 after expression: " + iOb2);

      // The same expression is evaluated, but the
      // result is not reboxed. i = iOb + (iOb
      / 3);
      System.out.println("i after expression: " + i);

   }
}
```

The output is shown here:

```
Original value of iOb: 100 After
++iOb: 101
iOb2 after expression: 134 i after
expression: 134
```

In the program, pay special attention to this line:

```
++iOb;
```

This causes the value in **iOb** to be incremented. It works like this: **iOb** is unboxed, the value is incremented, and the result is reboxed.

Auto-unboxing also allows you to mix different types of numeric objects in an expression. Once the values are unboxed, the standard type promotions and conversions are applied. For example, the following program is perfectly valid:

```
class AutoBox4 {
   public static void main(String[] args) {

      Integer iOb = 100; Double dOb
      = 98.6;

      dOb = dOb + iOb;
      System.out.println("dOb after expression: " + dOb);
   }
}
```

The output is shown here:

```
dOb after expression: 198.6
```

As you can see, both the **Double** object **dOb** and the **Integer** object **iOb** participated in the addition, and the result was reboxed and stored in **dOb**.

Because of auto-unboxing, you can use **Integer** numeric objects to control a **switch** statement. For example, consider this fragment:

```
Integer iOb = 2; switch(iOb) {
   case 1: System.out.println("one"); break;
   case 2: System.out.println("two"); break;
   default: System.out.println("error");
}
```

When the **switch** expression is evaluated, **iOb** is unboxed and its **int** value is obtained.

As the examples in the programs show, because of autoboxing/unboxing, using numeric objects in an expression is both intuitive and easy. In the early days of Java, such code would have involved casts and calls to methods such as **intValue( )**.

Autoboxing/Unboxing Boolean and Character Values

Java also supplies wrappers for **boolean** and **char**. These are **Boolean** and **Character**. Autoboxing/unboxing applies to these wrappers, too. For example, consider the following program:

```
// Autoboxing/unboxing a Boolean and Character.

class AutoBox5 {
   public static void main(String[] args) {


            // Autobox/unbox a boolean. Boolean b =
            true;

            // Below, b is auto-unboxed when used in
            // a conditional expression, such as an if. if(b)
            System.out.println("b is true");

            // Autobox/unbox a char. Character ch = 'x'; //
            box a char char ch2 = ch; // unbox a char

            System.out.println("ch2 is " + ch2);
         }
      }
```

The output is shown here:

```
b is true ch2 is
x
```

Autoboxing/Unboxing Helps Prevent Errors

In addition to the convenience that it offers, autoboxing/unboxing can also help prevent errors. For example, consider the following program:

```
// An error produced by manual unboxing. class
UnboxingError {
   public static void main(String[] args) {

      Integer iOb = 1000; // autobox the value 1000

      int i = iOb.byteValue(); // manually unbox as byte !!!

      System.out.println(i);   // does not display 1000 !
   }
}
```