18-Enum

Author: Vincent Lau

Learning Objectives

> Understand the purpose and benefits of enum types
>
> Define an enum with constructors, fields and methods
>
> Understand enum types in switch/ loop/ conditioning
>
> Basic methods & structures to construct a enum class

Overview

- The ***enum*** keyword was first introduced in Java 5.
- *enum* denotes a special type of class that **always internally extends *java.lang.Enum* class, so enum class cannot extend to any other class.** An *enum type* is a special data type that enables a variable to be a set of predefined constants.

- Common examples include compass directions (values of NORTH, SOUTH, EAST and WEST), and the days of the week (MONDAY, TUESDAY...). Because they are constants, the names of an enum type's fields are in uppercase letters.

```
public enum Currency {
    USD, GBP, EUR, HKD
}
```

Built-in Methods

- ***enums*** come with **built-in useful methods, like values(), name(), valueOf(), ordinal(), etc**. Otherwise, we need to write if we use the traditional public static final constants.
  - `values()` : Return the list of enum values defined in the enum class
  - `name()` : Return the String of enum value.
  - `valueOf()` : Usually used to **inversely check** enum value by the string value.
  - `ordinal()` : Return the order of enum instance.

Fields, Methods & Constructors

- We can define constructors, methods, and fields inside enum types.

```
enum Currency {
    USD("US Dollar", "Federal Reserve"),
    GBP("Pound Sterling", "Bank of England"),
    EUR("Euro", "European Central bank"),
    JPY("Japanese Yen", "Bank of Japan");

    private final String currName;
    private final String centralBankName;

    private Currency(String currName, String centralBankName) {
        this.currName = currName;
        this.centralBankName = centralBankName;
    }

    public String getCurrName() {
        return this.currName;
    }

```

```
    public String getCentralBankName() {
        return this.centralBankName;
    }

    @Override
    public String toString() {
        return String.format("Currency [currName=%s, centralBankName=%s]",
                this.currName, this.centralBankName);
        // return "Currency [currName=" + this.currName + ", centralBankName="
        //  + this.centralBankName + "]";
    }
}
```

Conditionals

- Since enum types ensure that only one instance of the constants exist in the JVM, we can safely use the "==" operator to compare two variables.

```
public class EnumDemo {
    public static void main(String[] args) {
        printCurrencyConditionally();
    }

    public static void printCurrencyConditionally() {
        for (Currency ccy : Currency.values()) {
            if (ccy == Currency.USD) {
                System.out.printf("%s has stopped Quantitative Easing!",
                        ccy.getCentralBankName());
            } else if (ccy == Currency.EUR) {
                System.out.printf("%s is fluctuating!\n", ccy.valueOf("EUR"));
            }
        }
    }
}

/*
Output:
Federal Reserve has stopped Quantitative Easing!
EUR is fluctuating!
*/
```

Access enum

For-Each Loop

```
1  public class EnumDemo {
2
3      public static void main(String[] args) {
4          printCurrencies();
5      }
6
7      public static void printCurrencies() {
8          for (Currency ccy : Currency.values()) { // Loop and list all Currency
9              System.out.println(ccy);
10         }
11     }
12 }
13 /*
14 Output:
15
16 Currency [name=US Dollar, centralBankName=Federal Reserve]
17 Currency [name=Pound Sterling, centralBankName=Bank of England]
18 Currency [name=Euro, centralBankName=European Central bank]
19 Currency [name=Japanese Yen, centralBankName=Bank of Japan]
20 */
```

Switch Statements

- Due to type-safe, it is safe to use switch statements to implement logic. Avoid unexpected input parameters passing in the switch statement.

```
1  public class EnumDemo {
2      public static void main(String[] args) {
3          useEnumTypesInSwitch();
4      }
5
6      public static void useEnumTypesInSwitch() {
7          for (Currency ccy : Currency.values()) { // Loop and list all Currency
8              switch (ccy) { // enum
9                  case EUR:
10                     System.out.println(ccy.currName() + "(" + ccy.name()
11                             + ")" + " is fluctuating!");
```

```
                    break;
                case USD:
                    System.out.println(ccy.currName() + "(" + ccy.name()
                            + ")" + " is rising!");
                    break;
                case GBP:
                    System.out.println(ccy.currName() + "(" + ccy.name()
                            + ")" + " is dropping!");
                    break;
                case JPY:
                    System.out.println(ccy.currName() + "(" + ccy.name()
                            + ")" + " is stagnant...");
                    break;
                default: throw new IllegalArgumentException("Unknown
currency");
            }
        }
    }
}

/*
Output:

US Dollar(USD) is rising!
Pound Sterling(GBP) is dropping!
Euro(EUR) is fluctuating!
Japanese Yen(JPY) is stagnant...
*/
```

Why do we need variable in enum class?

- Sometimes a special value can represent for a specific enum type.
- Sometimes we can use the associated value of an enum value for DB storage. (i.e. We can store 1 for Monday; 2 for Tuesday, etc)
- Make use of Constructor to achieve value association.

```
public enum WeekDays {
    MONDAY(1), TUESDAY(2), WEDNESDAY(3),
    THURSDAY(4), FRIDAY(5), SATURDAY(6),
    SUNDAY(7);

    private int dayNumber;

```

```
    private WeekDays(int dayNumber) {
        this.dayNumber = dayNumber;
    }
}
```

Benefits of enum

- While we can define constants in the traditional "*public static final*" way, it is preferred to **group related constants together** in enums. Constants defined this way have multiple benefits:
  - Make the code more readable
  - Allow for compile-time checking
  - Document the list of accepted values upfront
  - Avoid unexpected behavior due to invalid values being passed in (**type-safe**)

The World without enum

- Without enum, we probably would use "public static final"
- If we write a method with one paraement *Season without enum*, it is likely you have to implement parameter checking yourself and validate during runtime; if the parameter is enum type, then compiler would complain when invalid parameter is passed, we call it *type-safe*.
- The classes Season2 & Season3 are hard for developers to use, it is easy to create bugs during runtime.

```
public enum Season { // impilicit static final, why?
    SPRING(1), SUMMER(2), AUTUMN(3), WINTER(4);

    private int code;

    private Season(int code) {
        this.code = code;
    }
    // other methods ....
}

// Both Season2 & Season3 hard to check invalid value
public class Season2 {
    public static final String SPRING = "Spring";
    public static final String SUMMER = "Summer";
```

```
    public static final String AUTUMN = "Autumn";
    public static final String WINTER = "Winter";
}

// Low readability
public class Season3 {
    public static final int SPRING = 0;
    public static final int SUMMER = 1;
    public static final int AUTUMN = 2;
    public static final int WINTER = 3;
}
```

More Examples

- Directions

```
enum Direction {
  NORTH(1, 'N', "it is North"),
  EAST(2, 'E', "it is East"),
  SOUTH(-1, 'S', "it is South"),
  WEST(-2, 'W', "it is West");

  private int code;
  private char dbValue;
  private String description;

  private Direction(int code, char dbValue, String description) {
    this.code = code;
    this.dbValue = dbValue;
    this.description = description;
  }

  public int getCode() {
    return this.code;
  }

  public char getDbValue() {
    return this.dbValue;
  }

  public String getDescription() {
    return this.description;
  }

  public Direction opposite() {
```

```
    return Direction.get(this.code * -1);
  }

  public static boolean isOpposite(Direction d1, Direction d2) {
    return d1.getCode() * -1 == d2.getCode();
  }

  public static Direction get(int code) { // overloading
    for (Direction d : values()) {
      if (d.code == code)
        return d;
    }
    // return null;
    throw new IllegalArgumentException(
        "Cannot parse into an element of Direction by code : '" + code + "'");
  }

  public static Direction get(char dbValue) { // overloading
    for (Direction d : values()) {
      if (d.dbValue == dbValue)
        return d;
    }
    // return null;
    throw new IllegalArgumentException(
        "Cannot parse into an element of Direction dbValue: '" + dbValue +
"'");
  }
}

public class DemoEnum {
  public static void main(String[] args) {
    System.out.println(Direction.get('N')); // print NORTH
    System.out.println(Direction.get(-1)); // print SOUTH
    System.out.println(Direction.WEST.opposite()); // print EAST
    System.out.println(Direction.isOpposite(Direction.SOUTH,
Direction.NORTH)); // print true
    System.out.println(Direction.isOpposite(Direction.SOUTH, Direction.EAST));
// print false
    System.out.println(Direction.valueOf("NORTH")); // print enum
Direction.NORTH
    System.out.println(Direction.NORTH); // print enum Direction.NORTH
    System.out.println(Direction.NORTH.name()); // print enum Direction.NORTH
  }
}
```

- Status - often used to represent transaction or process status, and its code will be stored in database.

```
enum Status {
    ORDERED(0),
    PAID(1),
    SHIPPED(2),
    COMPLETED(3);

    private int code;

    private Status(int code) {
        this.code = code;
    }

    public int getCode() {
        return this.code;
    }

    public static boolean isForwardStatus(Status oldStatus, Status newStatus) {
        return newStatus.getCode > oldStatus.getCode;
    }

    public static Status get(int code) {
        for(Status s : values()) {
            if(s.code == code) return s;
        }
        return null; // or you can throw IAE
    }
}
```

Questions

- Why do we use enum types? What are the benefits?
- Define an enum for the days of the week. The enum should have an *integer* field that indicates the numeric value of the day (e.g. 1 for Monday and 2 for Tuesday), and the corresponding getter method. Override the toString() method as well.