Java SE & Spring

Module 1: Java SE

10.Enums & Widely Used Java APIs

Enum

What is Enum?

Java enum, also called Java enumeration type, is a type whose fields consist of a fixed set of constants. The very purpose of **enum** is to enforce compile time type safety. **enum** keyword is reserved keyword in Java.

We should use **enum** when we know all possible values of a variable at compile time or design time, though we can add more values in future as and when we identify them. In this java enum tutorial, we will learn what enums are and what problems they solve?

Enum

Suppose, we need directions in our program, so we can define constant values for every direction:

```java
public static final int START = 1;
public static final int WAITING = 2;
public static final int RUNNING = 3;
public static final int DEAD = 4;
```

There are potential risks for this approach. If we change the constant value, the client side(users of our constants) may not be aware of this change, because they probably will not get any error on their side. But the result that they received may not be the desired one. Let’s go to the Intellij (chapter 10) and figure out this case.

The other problem is, when we want to display the constant, we can only access the value of the constant, but not the value name.

Enum

The solution of this risks and problems is enums. If we change the enum value, the clients of our problem will get an error, because the old value is no longer available.

We can create enums both as a member of class or a standalone enum class, each defining case, we can define a public enum member like this:

```
private enum MeetingAcceptance {
        ACCEPT,
        DENY,
        TENTATIVE
}
```

We can use **MeetingAcceptance** enum like this:

```
MeetingAcceptance  accept = MeetingAcceptance.ACCEPT;
        System.out.println(accept);
```

Enum Methods

- **ordinal()** : returns the position of an enum constant.
- **compareTo()** : method compares the enum constants based on their ordinal value. Result is an int value.
- **toString()** : returns the string representation of the enum constants.
- **name()** : returns the defined name of an enum constant in string form.
- **valueOf()** : takes a string and returns an enum constant having the same string name.
- **values()** : returns an array of enum type containing all the enum constants.

Enum Constructor

An **enum** class may include a constructor like a regular class. These enum constructors are either

- **private** - accessible within the class

or

- **package-private (default)** - accessible within the package

Example:

```java
// enum fields
    EAST(0), WEST(180), NORTH(90), SOUTH(270);

    private Direction(final int angle) {
        this.angle = angle;
    }

    private int angle;

    public int getAngle() {
        return angle;
    }
```

DateTime API

Date and Time APIs

Java supports date and time features using primarily two packages **java.time** and **java.util**. The package **java.time** is added in Java 8 (JSR-310), and new added classes have aimed to solve the shortcomings of the legacy java.util.Date and java.util.Calendar classes.

Legacy Date Time API

The primary classes before Java 8 release were :

- **System.currentTimeMillis()** : represents the current date and time as milliseconds since January 1st 1970.
- **java.util.Date** : represents a specific instant in time, with millisecond precision.
- **java.util.Calendar** : an abstract class that provides methods for converting between instances and manipulating the calendar fields in different ways.
- **java.text.SimpleDateFormat** : a concrete class for formatting and parsing dates in a locale-sensitive manner and any predefined as well as any user-defined pattern.
- **java.util.TimeZone** : represents a time zone offset, and also figures out daylight savings.

Challenges of Legacy Date Time API

Though these APIs served the simple use cases very much, still Java community continuously complained about the problems in using these classes in an effective manner. For this reason, many other 3rd party libraries (e.g. Joda-Time or classes in Apache Commons) were more popular.

- A Date class shall represent a date, but it represents an instance which has hour, minutes and seconds as well.
- <u>But Date doesn't have any associated time zone.</u> It picks up the default timezone automatically. You cannot represent a date some other timezone.
- Classes are mutable. So that leaves additional burden on developers to clone the date before passing a function, which can mutate it.

Challenges of Legacy Date Time API

- Date formatting classes are also not thread safe. A formatter instance cannot be used without additional synchronization, else code may break.
- For some reason, there is another class java.sql.Date which has timzone information.
- Creating a date with some other timezone is very tricky and often result in incorrect result.
- Its classes use a zero-index for months, which is a cause of many bugs in applications over the years.

New Date Time API in Java 8

The new date api tries to fix above problems with legacy classes. It contains mainly following classes:

- **java.time.LocalDate** : represents a year-month-day in the ISO calendar and is useful for representing a date without a time. It can be used to represent a date only information such as a birth date or wedding date.
- **java.time.LocalTime** : deals in time only. It is useful for representing human-based time of day, such as movie times, or the opening and closing times of the local library.
- **java.time.LocalDateTime** : handles both date and time, without a time zone. It is a combination of LocalDate with LocalTime.
- **java.time.ZonedDateTime** : combines the LocalDateTime class with the zone information given in ZoneId class. It represent a complete date time stamp along with timezone information.

New Date Time API in Java 8

- **java.time.Duration** : Differnce between two instants and measured in seconds or nanoseconds and does not use date-based constructs such as years, months, and days, though the class provides methods that convert to days, hours, and minutes.
- **java.time.Period** : To define the difference between dates in date-based values (years, months, days).
- **java.time.ZoneId** : specifies a time zone identifier and provides rules for converting between an Instant and a LocalDateTime.
- **java.time.ZoneOffset** : specifies a time zone offset from Greenwich/UTC time.
- **java.time.format.DateTimeFormatter** : provides numerous predefined formatters, or we can define our own. It provides parse() or format() method to parsing and formatting the date time values.

Strings & Number Formatter

String in Java

A Java String represents an immutable sequence of characters and cannot be changed once created.

Strings are of type java.lang.String class. There are two ways to create a String in Java:

1. <u>**String Literal**</u>

   String literals are most easy and recommended way to create strings in Java. In this way, simply assign the characters in double quotes to variable of java.lang.String type.

```java
String healthTip = "You should drink at least 2 liter water daily!";

String welcomeMessage = "Hello World !!";
```

String literals are stored in String pool, a special memory area created by JVM. There can be only one instance of one String. Any second String with same character sequence will have the reference of first string stored in string pool. It makes efficient to work with Strings and saves lots of physical memory in runtime.

String in Java

```java
String healthTip1 = "You should drink at least 2 liter water daily!";
String healthTip2 = "You should drink at least 2 liter water daily!";
String healthTip3 = "You should drink at least 2 liter water daily!";
String healthTip4 = "You should drink at least 2 liter water daily!";
String healthTip5 = "You should drink at least 2 liter water daily!";
```

In above example, we created 5 string literals with same char sequence. Inside JVM, there will be only one instance of String inside string pool. All rest 4 instances will share the reference of string literal created for first literal.

String in Java

2. **String Object**

At times, we may wish to create separate instance for each separate string in memory. We can create one string object per string value using new keyword.

```java
String healthTip1 = new String("You should drink at least 2 liter water daily!");
String healthTip2 = new String("You should drink at least 2 liter water daily!");
```

In above example, there will be 2 separate instances of String with same value in heap memory.

String Methods

- **char charAt(int index)** - Returns the character at the specified index. Specified index value should be between '0' to 'length() -1' both inclusive.
- **boolean equals(Object obj)** - Compares the string with the specified string and returns true if both matches else false.
- **boolean equalsIgnoreCase(String string)** - Compares same as equals method but in case insensitive way.
- **int indexOf(int ch)** - Returns the index of first occurrence of the specified character argument in the string.
- **int indexOf(int ch, int fromIndex)** – Overloaded version of indexOf(char ch) method however it starts searching in the string from the specified fromIndex.

String Methods

- <u>**int lastIndexOf(int ch)**</u> – Returns the last occurrence of the character 'ch' in the string.
- **String substring(int beginIndex)** - Returns the substring of the string. The substring starts with the character at the specified index.
- **String concat(String str)** – Concatenates the specified string argument at the end of the string.
- **String toUpperCase()** - Converts the string to upper case string.
- **String toLowerCase()** - Converts the string to lower case string.
- **boolean isEmpty()** – Returns true if the given string has 0 length else returns false.

Java 8 - 11 String Methods

Java 8:

- **String join()** - A new static method join() has been added in String class in Java 8. This method returns a new String composed of copies of the CharSequence elements joined together with a copy of the specified delimiter.

Java 11:

- **isBlank()** – returns true if the string is empty or contains only white space codepoints, otherwise false.
- **lines()** – returns a stream of lines extracted from this string, separated by line terminators.
- **strip(), stripLeading(), stripTrailing()** – for stripping leading and trailing white spaces from the string.
- **repeat()** – returns a string whose value is the concatenation of this string repeated given number of times.

Number Formatting

We can print out numbers by using printf() or format() methods.

Example:

```java
long n = 461012;
System.out.format("%d%n", n);        //  -->  "461012"
System.out.format("%08d%n", n);      //  -->  "00461012"
System.out.format("%+8d%n", n);      //  -->  " +461012"
System.out.format("%,8d%n", n);      // -->  " 461,012"
System.out.format("%+,8d%n%n", n); //  -->  "+461,012"

double pi = Math.PI;

System.out.format("%f%n", pi);         // -->  "3.141593"
System.out.format("%.3f%n", pi);       // -->  "3.142"
System.out.format("%10.3f%n", pi);    // -->  "      3.142"
System.out.format("%-10.3f%n", pi);   // -->  "3.142"
System.out.format(Locale.FRENCH,"%-10.5f%n%n", pi); // -->  "3,14159   "
System.out.printf("%.4f%n", pi); // -->3.1416

```

Number Formatting

You can find details about the flags like , %f, %d, %n etc in the official Oracle Java documentation:

https://docs.oracle.com/javase/tutorial/java/data/numberformat.html#:~:text=follows%20the%20table.-,Co

nverters%20and%20Flags%20Used%20in%20TestFormat.java,-Converter

Number Formatting

The other way to formatting numbers is using **DecimalFormat** class. You can use the **java.text.DecimalFormat** class to control the display of leading and trailing zeros, prefixes and suffixes, grouping (thousands) separators, and the decimal separator. DecimalFormat offers a great deal of flexibility in the formatting of numbers, but it can make your code more complex.

Example:

```java
public static void customFormat(String pattern , double value ) {
        DecimalFormat myFormatter =  new DecimalFormat(pattern);
        String output = myFormatter.format(value);
        System.out.println(value + "   " + pattern + "   " + output);
    }
    public static void main(String[] args) {
        customFormat("###,###.###", 123456.789);
        customFormat("###.##", 123456.789);
        customFormat("000000.000", 123.78);
        customFormat("$###,###.###", 12345.67);
    }
```

Questions ?

Next:Lambda Expressions, Functional Interfaces & Stream API