Object Oriented Programing 1

Lecture 13: Java Enums, Java User Input & Java Date

By

Elubu Joseph

MSci.IS

Email: josebulinda@gmail.com

or

jose@kumiuniversity.ac.ug

Agenda

1. Java Enums,
2. Java User Input & Output stream
3. Java Date

Java Enums

Java Enums

The **Enum in Java** is a data type which contains a fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc.

According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

Java Enums

Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change).

Enums are used to create our own data type like classes. The **enum** data type (also known as Enumerated Data Type) is used to define an enum in Java. Unlike C/C++, enum in Java is more *powerful*. Here, we can define an enum **either inside** the class **or outside the class**.

Java Enum internally inherits the *Enum class*, so it cannot inherit any other class, but it can implement many interfaces. We can have fields, constructors, methods, and main methods in Java enum.

Remember that Java Enum

1. Enum improves type safety
2. Enum can be easily used in switch
3. Enum can be traversed
4. Enum can have fields, constructors and methods
5. Enum may implement many interfaces but cannot extend any class because it internally extends Enum class

Simple Example of Java Enum

```
class enumPro{
//defining the enum inside the class
public enum Season {   SUMMER, SPRING,  WINTER, FALL }

    public static void main(String[] args) {
    //traversing the enum
        for (Season s : Season.values())
        System.out.println(s);
    }

}
```

```
run:
SUMMER
SPRING
WINTER
FALL
```

Let us see another example of Java enum where we are using value(), valueOf(), and ordinal() methods of Java enum.

```
class EnumPro1{
      //defining enum within class
      public enum  Months  { JAN, FEB, MAR, APR,MAY,JUNE,JULY,AUG,SETP,OCT }
      //creating the main method
            public static void main(String[] args) {
            //printing all enum
            for (Months  m :  Months.values()){
                  System.out.print(m+", ");
            }
System.out.println("\nThe Second Month is: "+ Months.valueOf("FEB"));
System.out.println("Index of FEB is: "+ Months.valueOf("FEB").ordinal());
System.out.println("Index of JUNE is: "+ Months.valueOf("JUNE").ordinal());
  }
}
```

OUTPUT

```
run:
JAN, FEB, MAR, APR, MAY, JUNE, JULY, AUG, SETP, OCT,
Second Month is: FEB
Index of FEB is: 1
Index of JUNE is: 5
```

Note: Java compiler internally adds values(), valueOf() and ordinal() methods within the enum at compile time. It internally creates a static and final class for the enum.

Important questions to answer

1. **What is the purpose of the values() method in the enum?**

   The **values() method returns an array containing all the values** of the enum. The Java compiler internally adds the values() method when it creates an enum.

2. **What is the purpose of the valueOf() method in the enum?**

   The **valueOf() method** returns the value of given constant enum. The Java compiler internally adds the valueOf() method when it creates an enum.

3. **What is the purpose of the ordinal() method in the enum?**

   The **ordinal() method returns the index** of the enum value. The Java compiler internally adds the ordinal() method when it creates an enum.

Defining Java Enum

The enum can be defined within or outside the class because it is similar to a class. The semicolon (;) at the end of the enum constants are optional. For example:

1. **enum** Season { WINTER, SPRING, SUMMER, FALL }

Or,

2. **enum** Season { WINTER, SPRING, SUMMER, FALL**;** }

Both the definitions of Java enum are the same.

Java Enum Example: Defined outside class

```
enum Season { WINTER, SPRING, SUMMER, FALL }
class EnumOutPro{
    public static void main(String[] args){
        Season s=Season.WINTER;
         System.out.println(s.SPRING);
         System.out.println(s);

    }
}
```

OUTPUT

```
run:
SPRING
WINTER
```

Java Enum Example: Defined inside class

```
class enumPro2{
enum Season { WINTER, SPRING, SUMMER, FALL; }
//semicolon(;) is optional here
    public static void main(String[] args){

    Season s=Season.WINTER;//enum type is required t
o access WINTER
    System.out.println(s);
    }
}
```

```
WINTER
```

Java Enum Example: main method inside Enum

If you put main() method inside the enum, you can run the enum directly.

```
Public class mainInsideEnum {
enum Season{WINTER, SPRING, SUMMER, FALL;
public static void main(String[] args){
    Season s=Season.WINTER;
    System.out.println(s);
}
}}
```

OUTPUT

```
run:
WINTER
```

Initializing specific values to the enum constants

The enum constants have an initial value which starts from 0, 1, 2, 3, and so on. But, we can initialize the specific value to the enum constants by defining fields and constructors. As specified earlier, Enum can have fields, constructors, and methods.

Example of specifying initial value to the enum constants

Example of specifying initial value to the enum constants

```
class enumPro4{
      enum Season{  SUMMER(15), WINTER(5), SPRING(10),  FALL(20);

            private int value;
            private Season(int value){
                  this.value=value;
            }
      }
public static void main(String args[]){
      for (Season s : Season.values())
            System.out.println(s+" "+s.value);

}}
```

OUTPUT

```
run:
SUMMER 15
WINTER 5
SPRING 10
FALL 20
```

Some more questions.

1. Can we create the instance of Enum by new keyword?

No, because it contains private constructors only.

2. Can we have an abstract method in the Enum?

Yes, Of course! we can have abstract methods and can provide the implementation of these methods.

Java Enum in a switch statement

We can apply enum on switch statement as in the given example

```
class enumPro5{
enum Day{ SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY}
public static void main(String args[]){
        Day day = Day.SUNDAY;

        switch(day){
                case SUNDAY:
                        System.out.println("Honor the Creator");
                break;
                case MONDAY:
                        System.out.println("First day of the week");
                break;
                default:
                System.out.println("Working days");
        }
}}
```

```
run:
Honor the Creator
```

Java User Input/Output Stream

Java I/O (Input and Output)

Java I/O (Input and Output) is used *to process the input* and *produce the output*.

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in Java** by Java I/O API.

Stream

A stream is a sequence of data. In Java, a stream is composed of bytes. **It's called a stream because it is like a stream of water that continues to flow.**

The Java Streams

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

Let's see the code to print **output and an error** message to the console.

1. **System.out**.println("simple message");
2. System.err.println("error message");

Input Streams

Let's see the code to get **input** from console.

```
1.  int i=System.in.read();//returns ASCII code of 1st character
2.  System.out.println((char)i);//will print the character
```

Guiding Questions

Do You Know?

1. How can we write a common data to multiple files using a single stream only?
2. How can we access multiple files by a single stream?
3. How can we improve the performance of Input and Output operation?
4. How many ways can we read data from the keyboard?
5. What does the console class do?
6. How can we compress and uncompress the data of a file?

OutputStream vs InputStream

The explanation of OutputStream and InputStream classes are given below:

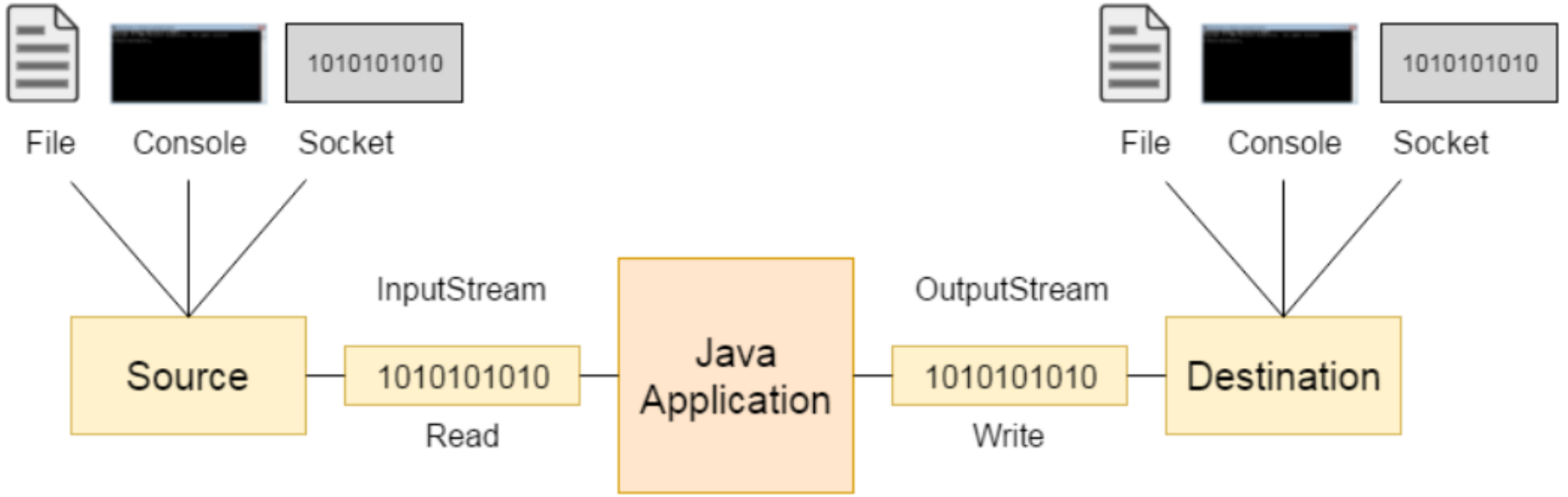
OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Let's understand the working of Java OutputStream and InputStream by the figure given below.



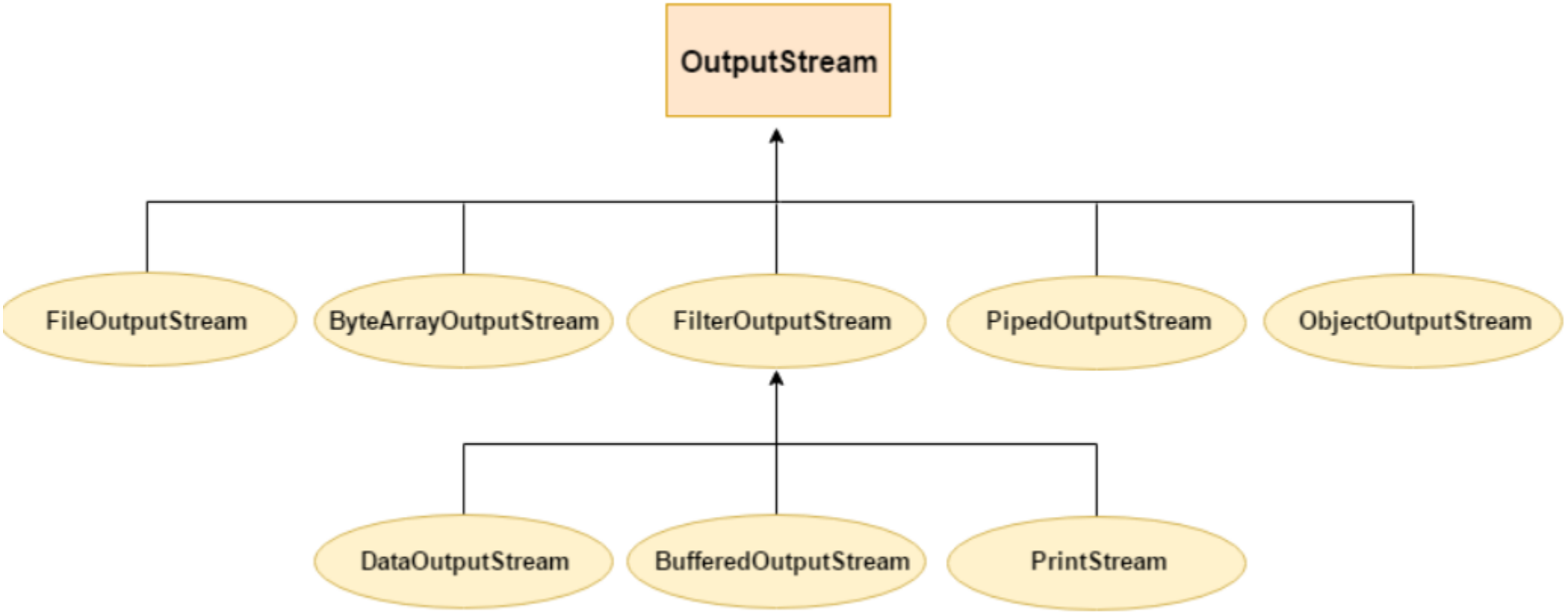
OutputStream class

OutputStream class is an abstract class. It is the super class of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Useful methods of OutputStream

Method	Description
1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

OutputStream Hierarchy



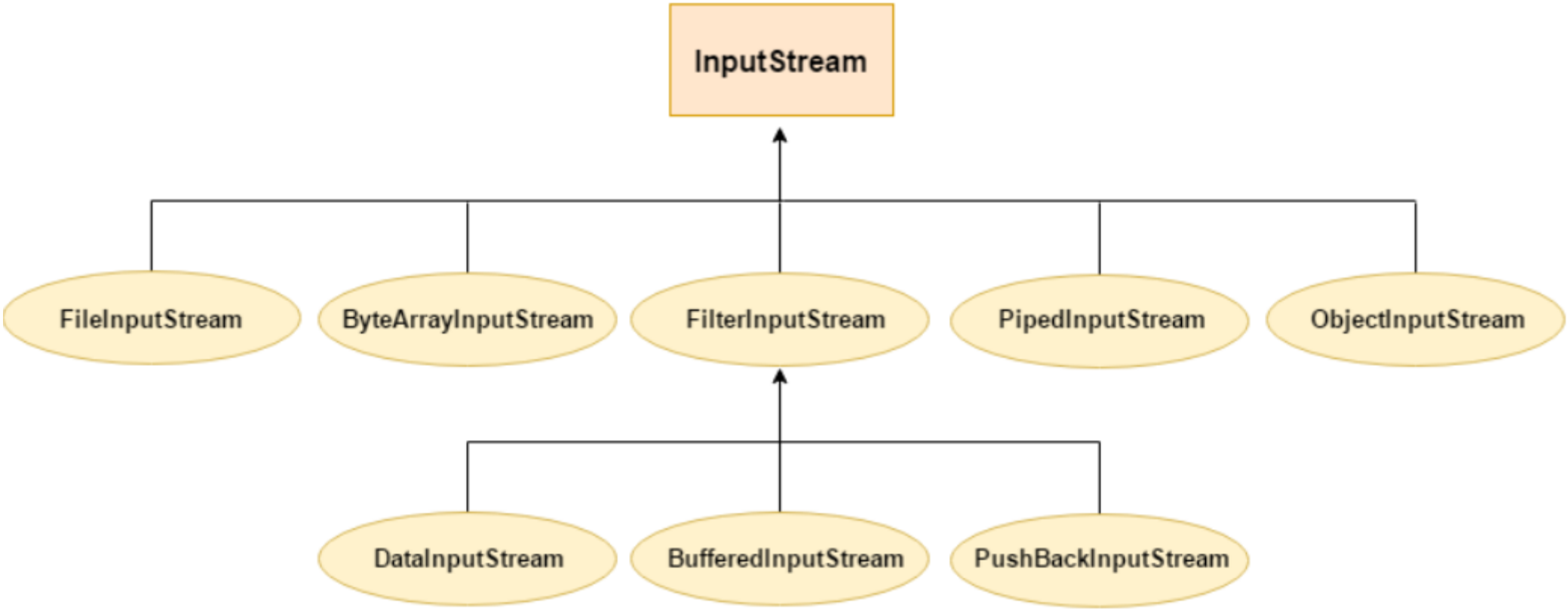
Input Stream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Useful methods of InputStream

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

InputStream Hierarchy



Java - Date and Time

Java - Date and Time

The java.time, java.util, java.sql and java.text packages contains classes for representing date and time. Following classes are important for dealing with date in Java.

Java 8 Date/Time API

Java has introduced a new Date and Time API since Java 8. The java.time package contains Java 8 Date and Time classes

Java 8 Date/Time API

- java.time.LocalDate class
- java.time.LocalTime class
- java.time.LocalDateTime class
- java.time.MonthDay class
- java.time.OffsetTime class
- java.time.OffsetDateTime class
- java.time.Clock class
- java.time.ZonedDateTime class
- java.time.ZoneId class
- java.time.ZoneOffset class
- java.time.Year class
- java.time.YearMonth class
- java.time.Period class
- java.time.Duration class
- java.time.Instant class
- java.time.DayOfWeek enum
- java.time.Month enum

Classical Date/Time API

But classical or old Java Date API is also useful. Let's see the list of classical Date and Time classes.

- java.util.Date class
- java.sql.Date class
- java.util.Calendar class
- java.util.GregorianCalendar class
- java.util.TimeZone class
- java.sql.Time class
- java.sql.Timestamp class

Formatting Date and Time

We can format date and time in Java by using the following classes:

- java.text.DateFormat class
- java.text.SimpleDateFormat class

Java Date and Time APIs

Java provide the date and time functionality with the help of two packages **java.time** and **java.util.** The package java.time is introduced in Java 8, and the newly introduced classes tries to overcome the shortcomings of the legacy java.util.Date and java.util.Calendar classes.

Classical Date Time API Classes

The primary classes before Java 8 release were:

1. **Java.lang.System:** The class provides the currentTimeMillis() method that returns the current time in milliseconds. It shows the current date and time in milliseconds from January 1st 1970.
2. **java.util.Date:** It is used to show specific instant of time, with unit of millisecond.

Classical Date Time API Classes

The primary classes before Java 8 release were:

3. **java.util.Calendar:** It is an abstract class that provides methods for converting between instances and manipulating the calendar fields in different ways.
4. **java.text.SimpleDateFormat:** It is a class that is used to format and parse the dates in a predefined manner or user defined pattern.
5. **java.util.TimeZone:** It represents a time zone offset, and also figures out daylight savings.

Drawbacks of the Old Date/Time API's

1. **Thread safety:** The existing classes such as Date and Calendar does not provide thread safety. Hence it leads to hard-to-debug concurrency issues that are needed to be taken care by developers. The new Date and Time APIs of Java 8 provide thread safety and are immutable, hence avoiding the concurrency issue from developers.
2. **Bad API designing:** The classic Date and Calendar APIs does not provide methods to perform basic day-to-day functionalities. The Date and Time classes introduced in Java 8 are ISO-centric and provides number of different methods for performing operations regarding date, time, duration and periods.
3. **Difficult time zone handling:** To handle the time-zone using classic Date and Calendar classes is difficult because the developers were supposed to write the logic for it. With the new APIs, the time-zone handling can be easily done with Local and ZonedDate/Time APIs.

New Date Time API in Java 8

The new date API helps to overcome the drawbacks mentioned above with the legacy classes. It includes the following classes:

1. **java.time.LocalDate:** It 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.
2. **java.time.LocalTime:** It 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.
3. **java.time.LocalDateTime:** It handles both date and time, without a time zone. It is a combination of LocalDate with LocalTime.

New Date Time API in Java 8+

The new date API helps to overcome the drawbacks mentioned above with the legacy classes. It includes the following classes:

4. **java.time.ZonedDateTime:** It combines the LocalDateTime class with the zone information given in ZoneId class. It represent a complete date time stamp along with timezone information.
5. **java.time.OffsetTime:** It handles time with a corresponding time zone offset from Greenwich/UTC, without a time zone ID.
6. **java.time.OffsetDateTime:** It handles a date and time with a corresponding time zone offset from Greenwich/UTC, without a time zone ID.

New Date Time API in Java 8++

The new date API helps to overcome the drawbacks mentioned above with the legacy classes. It includes the following classes:

7. **java.time.Clock :** It provides access to the current instant, date and time in any given time-zone. Although the use of the Clock class is optional, this feature allows us to test your code for other time zones, or by using a fixed clock, where time does not change.
8. **java.time.Instant :** It represents the start of a nanosecond on the timeline (since EPOCH) and useful for generating a timestamp to represent machine time. An instant that occurs before the epoch has a negative value, and an instant that occurs after the epoch has a positive value.

New Date Time API in Java 8+++

The new date API helps to overcome the drawbacks mentioned above with the legacy classes. It includes the following classes:

9. **java.time.Duration :** Difference 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.
10. **java.time.Period :** It is used to define the difference between dates in date-based values (years, months, days).
11. **java.time.ZoneId :** It states a time zone identifier and provides rules for converting between an Instant and a LocalDateTime.

New Date Time API in Java 8+++

The new date API helps to overcome the drawbacks mentioned above with the legacy classes. It includes the following classes:

12. **java.time.ZoneOffset :** It describe a time zone offset from Greenwich/UTC time.
13. **java.time.format.DateTimeFormatter :** It comes up with various predefined formatter, or we can define our own. It has parse() or format() method for parsing and formatting the date time values.

Java LocalDateTime class

Having seen anumber of date and time classes above, lets sample and dissect **LocalDateTime** class to use a practical eye.

Java LocalDateTime class is an immutable date-time object that represents a date-time, with the default format as yyyy-MM-dd-HH-mm-ss.zzz. It inherits object class and implements the ChronoLocalDateTime interface.

Java LocalDateTime class declaration

Let's see the declaration of java.time.LocalDateTime class.

public final class LocalDateTime **extends** Object **implements** Temporal, TemporalAdjuster, ChronoLocalDateTime<LocalDate>, Serializable

Lets see Methods of Java LocalDateTime

Methods of Java LocalDateTime

Method	Description
String format(DateTimeFormatter formatter)	It is used to format this date-time using the specified formatter.
int get(TemporalField field)	It is used to get the value of the specified field from this date-time as an int.
LocalDateTime minusDays(long days)	It is used to return a copy of this LocalDateTime with the specified number of days subtracted.
static LocalDateTime now()	It is used to obtain the current date-time from the system clock in the default time-zone.
static LocalDateTime of(LocalDate date, LocalTime time)	It is used to obtain an instance of LocalDateTime from a date and time.
LocalDateTime plusDays(long days)	It is used to return a copy of this LocalDateTime with the specified number of days added.
boolean equals(Object obj)	It is used to check if this date-time is equal to another date-time.

Java LocalDateTime Example program

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class LocalDateTimePro1 {
    public static void main(String[] args) {
        LocalDateTime now = LocalDateTime.now();
        System.out.println("Before Formatting: " + now);
          DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-
yyyy HH:mm:ss");
        String formatDateTime = now.format(format);
        System.out.println("After Formatting: " + formatDateTime);
    }
}
```

```
Before Formatting: 2022-06-09T16:14:13.636104900
After Formatting: 09-06-2022 16:14:13
```

Java LocalDateTime Example: now()

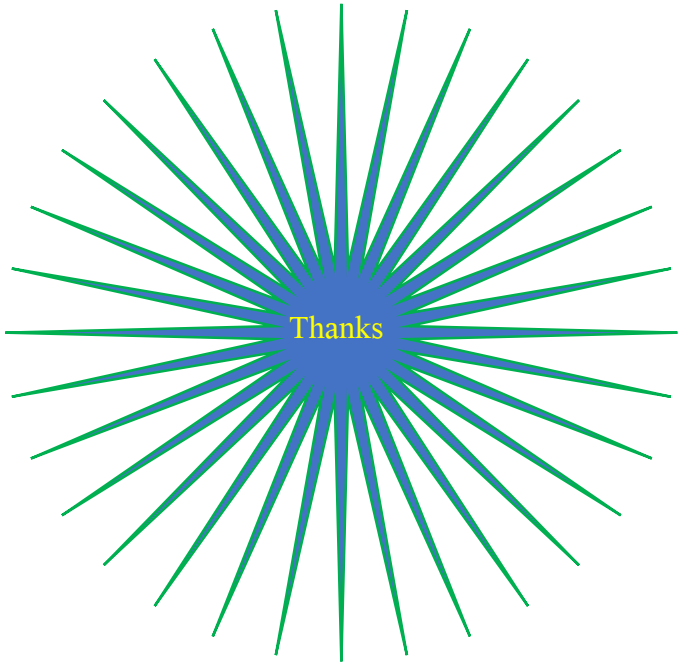
```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class LocalDateTimePro2 {
     public static void main(String[] args) {
    LocalDateTime datetime1 = LocalDateTime.now();
    DateTimeFormatter format = DateTimeFormatter.ofPattern("
dd-MM-yyyy HH:mm:ss");
    String formatDateTime = datetime1.format(format);
    System.out.println(formatDateTime);
  }
}
```

```
run:
09-06-2022 16:20:55
```

Summary

1. Java Enums (definition of enums,)
2. Java User Input &
3. Java Date

Thank you for Listening



Reference

Java enum - javatpoint. www.javatpoint.com. (n.d.). Retrieved June 1, 2022, from https://www.javatpoint.com/enum-in-java#:~:text=Java%20Enums%20can%20be%20thought,own%20data%20type%20like%20classes.

Java date and time - javatpoint. www.javatpoint.com. (n.d.). Retrieved June 7, 2022, from https://www.javatpoint.com/java-date