Day 34

## Using calendar Class

```
Calendar c = Calendar.getInstance();
c.setTime(yourDate);
int dayOfWeek = c.get(Calendar.DAY_OF_WEEK);
```

## Using SimpleDateFormat

```
String input_date="14/07/2020";
  SimpleDateFormat format1=new SimpleDateFormat("dd/MM/yyyy");
  Date dt1=format1.parse(input_date);
  DateFormat format2=new SimpleDateFormat("EEEE");
  String finalDay=format2.format(dt1);
```

## Using DateTimeFormatter

```
int dayOfWeek = localDate.getDayOfWeek();

Locale locale = Locale.US;

DateTimeFormatter formatterOutput = DateTimeFormat.forPattern( "E" ).withLocale( locale );

String output = formatterOutput.print( localDate );

String outputQuébécois =
formatterOutput.withLocale( Locale.CANADA_FRENCH ).print( localDate );


System.out.println( "input: " + input );

System.out.println( "localDate: " + localDate );

System.out.println( "dayOfWeek: " + dayOfWeek
```

## Get day in integer

```
import java.util.Calendar;
import java.util.Date;
import java.time.LocalDate;
import java.text.SimpleDateFormat;

public class LocalDateExample1 {
        public static void main(String[] args) {
                Date date = new Date();
                Calendar calendar = Calendar.getInstance();
                calendar.setTime(date);

                int year = calendar.get(Calendar.YEAR);
                int month = calendar.get(Calendar.MONTH);
                int day = calendar.get(Calendar.DAY_OF_MONTH);

                System.out.println("year : " + year);
                System.out.println("month : " + month);
                System.out.println("day : " + day);
        }
}
```

## Get DayOfWeek using the switch

```
import java.util.*;

public class DayOfWeek {

        public static void main(String[] args) {

                Calendar cal = Calendar.getInstance();
                int day = cal.get(Calendar.DAY_OF_WEEK);

                System.out.print("Today is : ");
                switch (day) {
                case 1:
                        System.out.print("Sunday");
                        break;
                case 2:
                        System.out.print("Monday");
                        break;
                case 3:
                        System.out.print("Tuesday");
                        break;
                case 4:
                        System.out.print("Wednesday");
                        break;
                case 5:
                        System.out.print("Thursday");
                        break;
                case 6:
                        System.out.print("Friday");
                        break;
                case 7:
                        System.out.print("Saturday");
                }

        }
}
```

## What is a stream?

- Well streams are an update the the Java API that let us manipulate collections of data in a declarative way. Declarative meaning that we tell the code what to do and not how to do it. However, if you want a more technical definition then here you go: `a stream is a sequence of elements from a source that supports data-processing operations`.
- Lets break this statement down a little:

**Sequence of elements :** a Stream provides an interface to a sequenced(preserving the order) set of values of a specific element type.

**Source :** Streams need to consume data from a data providing source (like an array)

**Data-processing operations :** Streams support database like operations and common operations from functional programming languages to manipulate data.

## What are streams really doing ?

- When talking about streams, you could talk and give examples all day long about the data processing operations they enable and the lazy loading optimizations. However, the big difference that makes streams so unique is `internal iteration`. Which means that traversing each element is handled for us internally by the stream library and we don't have to use a for loop. FYI using a for loop would be considered `external iteration`. Now, the only way for use to take advantage of `internal iteration` is if we are provided with predefined operations to work with. `.anyMatch()` is one such operation.

**anyMatch() :** this method is used to ask the question, is it true an element in the stream matches the given predicate.

- This definition might seem particularly unhelpful when we look at our `anyMatch` method: `s -> s.contains(path)`. To understand what is going on, we need to understand what a `Lambda expression` is;

## What is a Lambda expression?

- A lambda expression is a, `concise representation of an anonymous function that can be passed around`. Now that we have the technical jargon out of the way, lets break it down a bit:

**Anonymous :** we say anonymous because it doesn't have an explicit name like a method would

**Function :** we say function because a lambda isn't associated with a particular class like a method is.

**Passed around :** A lambda expression can be passed as an argument to a method or stored in a variable.

**Concise :** less code to write

- The term lambda actually comes from lambda calculus ( compsci nerds love lambda calculus)
- A typical lambda expression(like the one below) has three sections:

```
s -> s.contains(path)
```

**1) list of parameters**
**2) an arrow `->`**
**3) the body of the lambda**

- The key to really understanding and using a lambda expression is a `functional interface`

## Functional interface

- A functional interface is just an interface that specifies an abstract method, it could look something like this:

```
public interface Testing{
      boolean test(int test`, int test2);
}
```

Remember that an abstract method that is declared without an implementation(without braces and followed by a semicolon).

- How do lambdas and functional interfaces work together? A lambda expression lets us provide the implementation of the abstract method inside the functional interface. That might not make much sense now but I will elaborate more in the next section

## Type checking

- If we look as the lambdas that we made previously in this post, you will notice the lack of explicitly declaring types. Don't worry the types are not eliminated, the types used in a lambda are deduced from the context in which the lambda is used in . This type is called the `target type`. The context is the abstract method that this lambda is implementing.

## What is actually happening?

- So now lets try to break down what is actually happening inside the code for :

```
anyMatch(s -> s.contains(path))
```

**1)** The definition for anyMatch() is looked up

**2)** it expects an object of type Predicate (target type). You can read the documentation HERE, but the type is String because we are iterating over a stream of Strings

**3)** Predicate is a functional interface and has a single abstract method called test() documentation HERE

**4)** We then implement the test() method with our lambda expression. This means that our lambda expression gets is signature from the `test()` method. So our lambda will take in a single element and return a boolean.

**5)** We use `s.contains(path)` to check if the s matches the path and return the boolean.