Java Fundamentals - Day 2

Functional programming concepts

What is functional programming?

Why should someone use this programming paradigm?

The key concepts of functional programming:

- functions are the first-class citizens
- using pure functions
- immutable objects

Functional programming in the Java library

- **Lambda functions** - it’s a short way to create an instance of an interface with one abstract method. It’s possible to give them name and pass as a parameter to another function.
- **java.util.function** - a package with interfaces for every type of function.
- **java.util.stream** - a declarative way of working with data streams.
- **java.util.optional** - a class that helps to make a method signature more informative.

Lambda functions

```
BiFunction<String, Integer, String> simpleEncode =
                (s, offset) -> {
                    StringBuilder encoded = new StringBuilder();
                    for (int i = 0; i < s.length(); i++) {
                        encoded.append((char) (s.charAt(i) + offset));
                    }
                    return encoded.toString();
                };
```

The type of that function 'simpleEncode' is: `BiFunction<String, Integer, String>`, which means two arguments (`String` and `Integer`) function with return `String` value.

The basic syntax of lambda function are:

```
(argument1, argument2, ...) -> {
    body
}
```

```
IntFunction<Integer> twice = x -> 2 * x;
```

Rafactoring an example using lambda function as an argument.

```
public List<Integer> filterEven(List<Integer> list) {
        List<Integer> result = new ArrayList<>();
        for (var item : list) {
            if (item % 2 == 0) {
```

```
                result.add(item);
            }
        }
        return result;
    }
```

Refactored function:

```
public List<Integer> filterIntList(List<Integer> list, IntPredicate condition) {
        List<Integer> result = new ArrayList<>();
        for (var item : list) {
            if (condition.test(item)) {
                result.add(item);
            }
        }
        return result;
    }
```

Usage:

```
public List<Integer> filterEven(List<Integer> list) {
    return filterIntList(list, item -> item % 2 == 0);
}

public List<Integer> filterUneven(List<Integer> list) {
    return filterIntList(list, item -> item % 2 != 0);
}
```

Using method reference instead of lambda

```
Function<Integer, String> intToStringRef = Object::toString;
Function<Integer, String> intToStringLambda = n -> n.toString();
```

Functional interface

What is the functional interface?

Different types of functional interfaces in the standard library:

- **Function/BiFunction/IntFunction etc.** - has input, has output,
- **Consumer/BiConsumer/IntConsumer etc.** - has input, no output,
- **Supplier/IntSupplier** - no input, has output,
- **Predicate** - has input, boolean output.

Creating your own functional interface

```
@FunctionalInterface
public interface ThreeArgFunc<T, U, V, R> {
    R apply(T t, U u, V v);
}

ThreeArgFunc<String, String, String, String> myFunc = (x1, x2, x3) -> x1 + x2 + x3;
```

Assignment 01

Stream

What is the `Stream` class?

Properties:

- declarative
- lazy
- easy to read when you get used to it
- elements are visited once

Creating a stream

- **Stream.of** - creating stream from the passed arguments

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
```

- **Collection.stream** - produce a stream from collection

```
// Stream(1, 2, 3, 4)
Stream<Integer> stream = List.of(1, 2, 3, 4).stream();
```

- **Stream.generate** - produce an infinite sequential unordered stream where each element is generated by the passed supplier. This is suitable for

generating constant streams, streams of random elements, etc.

```
// Stream of 10 random numbers
RandomGenerator gen = RandomGenerator.getDefault();
Stream<Integer> stream = Stream.generate(() -> gen.nextInt(100))
    .limit(10);
```

- **Stream.iterate** - create a sequential ordered stream produced by iterative application of the passed function to an initial element, conditioned on satisfying the passed predicate.

```
// Stream(0, 1, 2, 3, 4)
Stream<Integer> stream = Stream.iterate(0, n -> n < 5, n -> n + 1);
```

Stream: intermediate operations

- **Stream.filter** - returns a stream consisting of the elements of this stream that match the given predicate

```
// Stream(2, 4)
Stream<Integer> evenStream = Stream.of(1, 2, 3, 4, 5)
    .filter(n -> n % 2 == 0)
```

- **Stream.map** - replace every element of the stream by another by applying the passed function

```
// Stream(2, 4, 6, 8, 10)
Stream<Integer> twiceStream = Stream.of(1, 2, 3, 4, 5)
    .map(n -> n * 2)
```

- **Stream.flatMap** - replace every element of the stream by different stream allowing to expand the initial one. Often used when it's needed to flatten the inner collection.

```
// Stream(1, 2, 3, 4, 5)
Stream<Integer> stream = Stream.of(List.of(1, 2), List.of(3, 4), List.of(5))
    .flatMap(Collection::stream);
```

- **Stream.limit** - limit elements in the stream

```
// Stream(1, 2)
Stream<Integer> limited = Stream.of(1, 2, 3, 4, 5).limit(2);
```

- **Stream.sorted** - sort the stream according to natural order or to the passed comparator

```
// Stream(2, 3, 5, 6, 54)
Stream<Integer> sorted = Stream.of(5, 2, 3, 6, 54).sorted();
```

Stream: terminal operations

- **Stream.forEach** - performs an action for each element of this stream. Order of iteration is nondeterministic.

```
Stream.of(5, 2, 3, 6, 54).forEach(System.out::println);
```

- **Stream.collect** - collect elements of the stream to a container.

```
// Before Java 16
List<Integer> list = Stream.of(5, 2, 3, 6, 54).collect(Collectors.toList());
```

```
// Since Java 16
List<Integer> list = Stream.of(5, 2, 3, 6, 54).toList();
```

- **Stream.match** - various matching operations can be used to check whether a given predicate matches the stream elements. All of these matching operations are terminal and return a boolean result.

```
List<String> names = List.of("Alex", "John", "Pablo");

// true
boolean res1 = names.stream().anyMatch(s -> s.startsWith("A"));

// false
boolean res2 = names.stream().allMatch(s -> s.startsWith("A"));

// false
boolean res3 = names.stream().noneMatch(s -> s.startsWith("A"));
```

- **Stream.count** - count elements in the stream

```
// 5
long count = Stream.of(5, 2, 3, 6, 54).count();
```

- **Stream.reduce** - performs a reduction on the elements of the stream with the given accumulation function

```
// 5 * 2 * 3 = 30
Integer res = Stream.of(5, 2, 3).reduce(1, (n, agg) -> n * agg);
```

- **Stream.findFirst / Stream.findAny** - returns optional of the first element if the initial stream is ordered, otherwise any element is returned

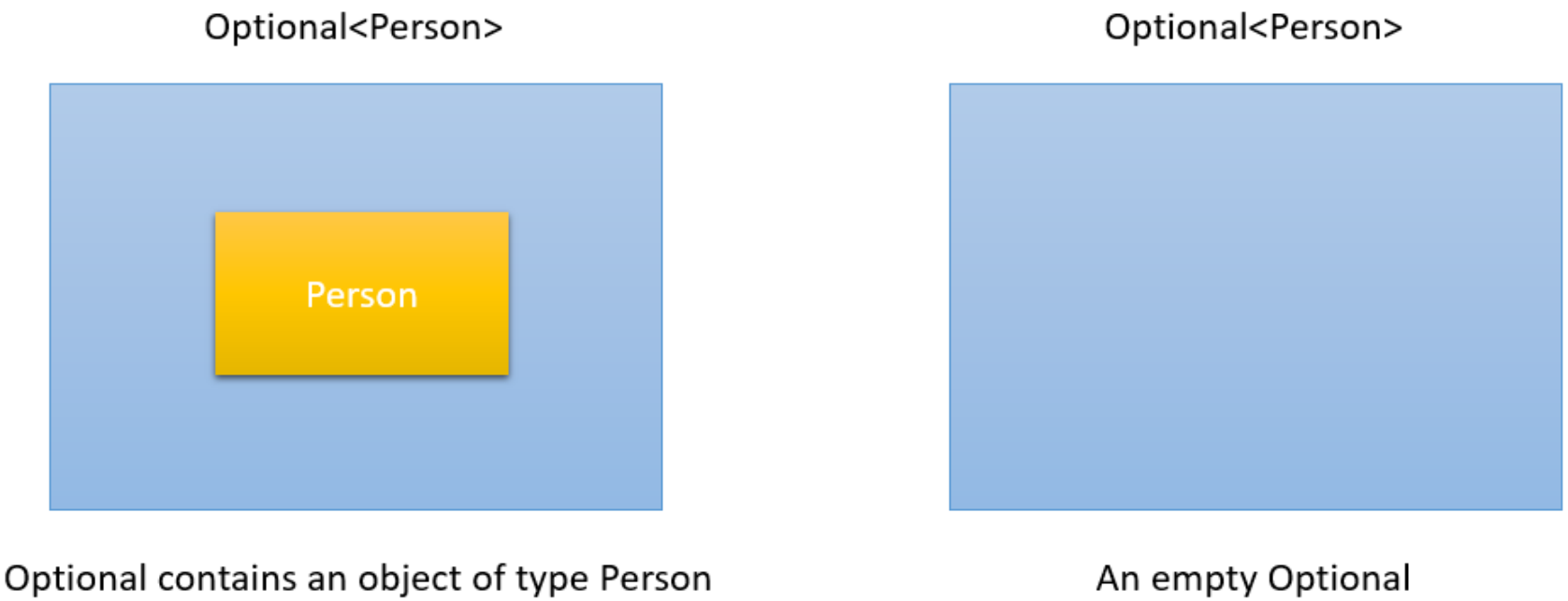
```
// Optional(5)
Optional<Integer> maybeNum = Stream.of(5, 2, 3, 6, 54).findFirst();

// Random number from stream
Optional<Integer> randomNum = Stream.of(5, 2, 3, 6, 54).findAny();
```

Assignment 02

Java Optional

What is Java Optional?



Solving the null problem with Optional

Using Optional

- Creation

```
Optional<String> maybeString = Optional.of("some string");
Optional<String> maybeString = Optional.ofNullabe(null);
Optional<String> maybeString = Optional.ofNullabe("some string");
```

- Using as improved method signature

Without Optional	With Optional
`User findUserById(long id);`	`Optional<User> findUserById(long id);`

- **Optional.orElse -** take the passed value if the optional is empty

Without Optional	With Optional

`User user = findUserById(long id);`
`if (user == null) {`
`  User user = new User();`
`}`
	`User user = findUserById(long id).orElse(new User());`

- **Optional.orElseThrow** - thrown an exception if the optional is empty

Without Optional	With Optional

`User user = findUserById(long id);`
`if (user == null) {`
`  thrown new UserNotFoundException();`
`}`
	`User user = findUserById(long id)`
	`  .orElseThrow(() -> new UserNotFoundException());`

- **Optional.ifPresentOrElse** - accepts two functions as input parameters: what to do with optional if it exists and what to do otherwise

Without Optional	With Optional

`User user = findUserById(long id);`
`if (user == null) {`
`  log.error("User is not found")`
`} else {`
`  doSomething(user);`
`}`
	`User user = findUserById(long id)`
	`  .ifPresentOrElse(`
	`  user -> doSomething(user),`
	`  () -> log.error("User is not found")`
	`  )`

- **Optional.map** - applies the function to the value inside the optional if the value is present, otherwise returns an empty optional.

Without Optional	With Optional

`String getUserStreet(long userId) {`
`  User user = findUserById(userId);`
`  if (user != null) {`
`  return user.getAddress().getStreet();`
`  }`
`  return null;`
`}`
	`Optional<String> getUserStreet(long userId) {`
	`  return findUserById(userId)`
	`  .map(User::getAddress)`
	`  .map(Address::getStreet);`
	`}`

- **Optional.filter** - matches the value with the passed predicate if value exists, otherwise returns an empty optional.

Without Optional	With Optional

```
String getUser(long userId) {
    User user = findUserById(userId);
    if (user != null && user.getAge() > 18) {
        return user;
    }
    return null;
}
```

```
Optional<String> getUser(long userId) {
    return findUserById(userId)
        .filter(user -> user.getAge() > 18);
}
```

- **Optional.flatMap** - applies the function to the value inside the optional if the value is present, otherwise returns an empty optional. The difference between map and flatMap is in the input mapping function signature: in flatMap the function must return an optional.

```
// Method User.getJobTitle returns Optional<String>, because not every user has a job title
findUserById(long userId) // Optional<User>
    .map(User::getJobTitle) // Optional<Optional<String>>

// This version is correct
findUserById(long userId) // Optional<User>
    .flatMap(User::getJobTitle) // <Optional<String>
}
```

Assignment 03

Date Time API

- Calendars are hard
- Timezones are hard
- Daylight savings is horrible (for devs)

- Introduce the trainees to the difficulties of maintaining dates and times. How countings seconds from 1-1-1970 seemed okay until 32 bits were full. How maintaining just the last two digits of the year seemed okay until 31-12-1999. How there is a Chinese calendard, a Gregorian calendar, an islamic calendar and probably a few more. Just imagine what happens when life goes interplanetary…

Date Time before Java 8

- **Thread safety**

The `java.util.Date` and `java.util.Calendar` are not thread safe. The **mutable** Date/Time classes lead developers to concurrency problems.

- **Inconsistent methods**

Poorly designed APIs those were not sufficient.

- **Zoned Date Time**

It needs additional code to handle time zone.

Date Time after Java 8

Java 8 introduced redesigned package `java.time`

- `LocalDate`
- `LocalTime`
- `LocalDateTime`
- `ZonedDateTime`

- `Period`
- `Duration`

instead of trying to put everything related to date/time into only a few classes with incomprehensible APIs, the new date/time API has many more classes, each with a clear name of what they represent. APIs that make sense to be shared between some of them are shared and the separation makes room for more specific APIs where those are needed. Dealing with timezones has become a lot clearer.

Immutability

Immutable Date/Time objects lead to improved software reliability and better multithreaded performance

Date-time classes in Java	Modern class	Legacy class
Moment in UTC	`java.time.Instant`	`java.util.Date` `java.sql.Timestamp`
Moment with offset-from-UTC (hours-minutes-seconds)	`java.time.OffsetDateTime`	(lacking)
Moment with time zone (`Continent/Region`)	`java.time.ZonedDateTime`	`java.util.GregorianCalendar`
Date & Time-of-day (no offset, no zone) **Not a moment**	`java.time.LocalDateTime`	(lacking)

joda-time is another project that wants to fill the date/time problem but after java 8 it advises to migrate to `java.time`

threeten-extra is a project that add some extra functionality to complete `java.time`

The base design decision underlying the new date time API is immutability. This makes all classes immediately thread safe and simply gives you less to worry about when passing Instants and LocalDates around.

Instant

Best practice to get the current moment in UTC with nanosecond resolution.

```
import java.time.*;

Instant instant = Instant.now();
System.out.println(instant);

//With ZoneId
System.out.println(instant.atZone(ZoneId.of("Europe/Istanbul")));
```

LocalDate

A date in ISO format (yyyy-MM-dd) without time. We can create in different ways, and it provides a wide variety of APIs to maintain it.

```
import java.time.*;
import java.time.temporal.*;

//Create With System date
System.out.println( LocalDate.now() );

//Create With Specific year, month, and day
```

```
System.out.println( LocalDate.of(2022, 07, 01) ) ;

//Create With Parsing a Date String
System.out.println( LocalDate.parse("2022-07-03") );
```

Calculus with dates is now easy and accessing the parts of a date is intuitive:

```
// APIs
System.out.println( LocalDate.now().plusDays(1) );

System.out.println( LocalDate.now().minus(1, ChronoUnit.MONTHS) );

System.out.println( LocalDate.now().getDayOfWeek() );

System.out.println( LocalDate.now().getDayOfMonth() );

System.out.println( LocalDate.now().isLeapYear() );

System.out.println( LocalDate.parse("2022-06-30").isBefore(LocalDate.parse("2022-07-01")) );

System.out.println( LocalDate.parse("2022-06-30").isAfter(LocalDate.parse("2022-07-01")) );

System.out.println( LocalDate.now().atStartOfDay() );

System.out.println( LocalDate.parse("2022-07-12").with(TemporalAdjusters.firstDayOfMonth()) );
```

LocalTime

Same as LocalDate for Time without date.

```
import java.time.*;
```

```
import java.time.temporal.*;

//Create With System date
System.out.println( LocalTime.now() );

//Create With Specific hour and minute
System.out.println( LocalTime.of( 17, 15) ) ;

//Create With Parsing a Time String
System.out.println( LocalTime.parse("17:15") );


// APIs
System.out.println( LocalTime.now().plus(1, ChronoUnit.HOURS) );

System.out.println( LocalTime.now().minus(1, ChronoUnit.HOURS) );

System.out.println( LocalTime.now().getHour() );

System.out.println( LocalTime.now().getMinute() );

System.out.println( LocalTime.now().isLeapYear() );

System.out.println( LocalTime.parse("16:30").isBefore(LocalTime.parse("17:00")) );

System.out.println( LocalTime.parse("16:30").isAfter(LocalTime.parse("17:00")) );

System.out.println( LocalTime.MAX );
```

LocalDateTime

Combination of Date and Time.

```
import java.time.*;
import java.time.temporal.*;

//Create With System date
System.out.println( LocalDateTime.now() );

//Create
System.out.println( LocalDateTime.of( 2022,Month.JULY,01,17, 15) ) ;

//Create With Parsing a Date Time String
System.out.println( LocalDateTime.parse("2022-07-010T17:15:00") );


// APIs
System.out.println(LocalDateTime.now().plusDays(1));

System.out.println( LocalDateTime.now().minusHours(2));

System.out.println( LocalDateTime.now().getMonth() );
```

ZonedDateTime

To maintain zone in data time, there are more than 40 zones that identify by the `ZoneId` in java 8.

Be careful about `ZonedDateTime.now()`. It will get the system default time zone, and it can change even in runtime.

```
import java.time.*;
import java.time.temporal.*;


System.out.println( ZoneId.getAvailableZoneIds() );
```

```
ZoneId zoneId = ZoneId.of("Europe/Istanbul");

LocalDateTime localDateTime = LocalDateTime.now();

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


//Zoned Date Time
System.out.println( "ZonedDateTime: " + ZonedDateTime.of(localDateTime, zoneId) );

//By parsing
System.out.println( ZonedDateTime.parse("2022-07-10T10:15:30+03:00[Europe/Istanbul]")) ;

//Create With `OffsetDateTime` give to specify costume offset on zoned date time.
ZoneOffset offset = ZoneOffset.of("+03:30");
System.out.println( OffsetDateTime.of(localDateTime, offset) );

//From Epoch
System.out.println( "From Epoch: " + LocalDateTime.ofEpochSecond(1565817690, 10, ZoneOffset.UTC));
```

Period and Duration

Period is used to show an amount of time in year, month, and day Duration is used to show an amount of time in hour or second

```
import java.time.*;
import java.time.temporal.*;


System.out.println( "==== PERIOD ====" );

// Period
LocalDate localDate = LocalDate.now();
```

```
System.out.println( localDate.plus(Period.ofDays(5)) );
System.out.println( localDate.plus(Period.ofMonths(1)) );
System.out.println( localDate.plus(Period.ofYears(1)) );

// To get the difference
LocalDate initializedLocalDate = LocalDate.now();
LocalDate finalLocalDate = initializedLocalDate.plus(Period.ofDays(37));

System.out.println( Period.between(initializedLocalDate, finalLocalDate).getDays());
System.out.println( Period.between(initializedLocalDate, finalLocalDate).getMonths());

System.out.println( "==== DURATION ====" );


// Duration
LocalTime localTime = LocalTime.now();

System.out.println( localTime.plus(Duration.ofSeconds(5)) );
System.out.println( localTime.plus(Duration.ofHours(1)) );

// To get the difference
LocalTime initializedLocalTime = LocalTime.now();
LocalTime finalLocalTime = initializedLocalTime.plus(Duration.ofSeconds(30).plus(Duration.ofMinutes(2)));

System.out.println( Duration.between(initializedLocalTime, finalLocalTime).getSeconds());
```

Java 8 added `toInstant` method to `Date` and `Calendar` class to make those compatible with new APIs.

```
import java.util.*;
import java.time.*;
import java.time.temporal.*;

System.out.println(LocalDateTime.ofInstant(new Date().toInstant(), ZoneId.systemDefault()));
```

Formatting

More easy formatters have been provided on java 8

```
import java.util.*;
import java.time.*;
import java.time.format.*;
import java.time.temporal.*;


System.out.println( LocalDateTime.now().format(DateTimeFormatter.ISO_DATE) );
System.out.println( LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy/MM/dd")) );
System.out.println(
LocalDateTime.now().format(DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM).withLocale(Locale.UK))
);
System.out.println(
LocalDateTime.now(ZoneId.of("Europe/Istanbul")).format(DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL
)) );
```