TRANSITION TECHNOLOGIES

JDK8 – Optionals and Date API

Using Optional as a better alternative to null

Using Optional as a better alternative to null

- NullPointerExceptions are a pain for any Java developer: novice or expert
- How to avoid unexpected NPE?

The Person/Car/Insurance data model

```
public class Person {
    private Car car;
    public Car getCar() { return car; }
}
public class Car {
    private Insurance insurance;
    public Insurance getInsurance() { return insurance; }
}
public class Insurance {
    private String name;
    public String getName() { return name; }
}
```

What is possibly problematic with the following code?

```
public String getCarInsuranceName(Person person) {
    return person.getCar().getInsurance().getName();
}
```

- **How to avoid NPE?**

Solution 1: Reducing NullPointerExceptions with defensive checking

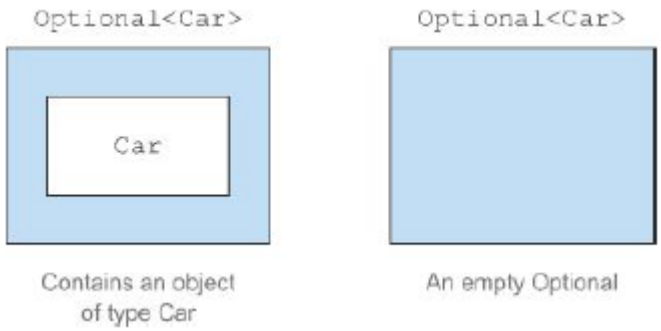
```
public String getCarInsuranceName(Person person) {

    if(person != null) {
        Car car = person.getCar();
        if(car != null) {
            Insurance insurance = car.getInsurance();
            if(insurance != null) {
                return insurance.getName();
            }
        }
    }

    return "Null somewhere...";
}
```

- **How to avoid NPE?**

Solution 2: Use OPTIONAL class from Java 8

java.util.Optional<T>

- Introduced in Java 8
- Inspired by the ideas of Haskell and Scala
- Wrapper
- A person might or might not have a car, the car variable inside the Person class should be type Optional<Car>



Redefining the Person/Car/Insurance data model using Optional

```
public class Person {
  private Optional<Car> car;
  public Optional<Car> getCar() { return car; }
}

public class Car {
  private Optional<Insurance> insurance;
  public Optional<Insurance> getInsurance() { return insurance; }
}

public class Insurance {
  private String name;
  public String getName() { return name; }
}
```

Creating Optional objects

- Empty optional
  - hold of an empty optional object using the static factory method *Optional.empty*

```
Optional<Car> optionalCar = Optional.empty();
```

- Optional from a non-null value
  - value with the static factory method *Optional.of*
  - if car were null, a NPE would be thrown

```
Optional<Car> optionalCar = Optional.of(car);
```

- Optional from null
  - Optional object that may hold a null value
  - If car were null, the resulting Optional object would be empty

```
Optional<Car> optionalCar = Optional.ofNullable(car);
```

Transforming values from optionals with map

- The map operation applies the provided function to each element of a stream.
- Let's think of an Optional object as a particular collection of data, containing at most a single element.
- If the Optional contains a value, then the function passed as argument to map transforms that value. If the Optional is empty, then nothing happens.

```
Insurance insurance1 = new Insurance();
Optional<Insurance> optInsurance1 = Optional.ofNullable(insurance1);
Optional<String> result1 = optInsurance1.map(Insurance::getName); //Optional.empty

Insurance insurance2 = new Insurance("AXA");
Optional<Insurance> optInsurance2 = Optional.ofNullable(insurance2);
Optional<String> result2 = optInsurance2.map(Insurance::getName); //Optional[AXA]
String result3 = optInsurance2.map(Insurance::getName).orElse("Temporary name"); //AXA

Insurance insurance3 = new Insurance();
Optional<Insurance> optInsurance3 = Optional.ofNullable(insurance3);
String result4 = optInsurance3.map(Insurance::getName).orElse("Temporary name");
                                    //Temporary name
```

Chaining Optional objects with flatMap

```
public String getCarInsuranceName() {

 Optional<Person> optPerson = Optional.empty();
 Optional<Optional<Car>> optOptCar = optPerson.map(Person::getCar);

 Optional<Car> optCar = optPerson.flatMap(Person::getCar);
 Optional<Insurance> optInsurance = optCar.flatMap(Car::getInsurance);


 return optInsurance.map(Insurance::getName).orElse("No car insurance name");
}
```

Chaining Optional objects with flatMap

```
public String getCarInsuranceName(Optional<Person> person) {
    return person.flatMap(Person::getCar)
                 .flatMap(Car::getInsurance)
                 .map(Insurance::getName)
                 .orElse("No car insurance name");
}
```

Summary

- Null-references have been historically introduced in programming languages to generally signal the absence of a value.
- Java 8 introduces the class java.util.Optional<T> to model the presence or absence of a value.
- You can create Optional objects with the static factory methods **Optional.empty**, **Optional.of**, and **Optional.ofNullable**.
- Using Optional can help you design better APIs in which, just by reading the signature of a method, users can tell whether to expect an optional value.

New Date and Time API in Java 8

New Date and Time API in Java 8

- Representing date and time for both humans and machines
- Defining an amount of time
  - LocalDate, LocalTime, Instant, Duration, and Period
- Manipulating, formatting, and parsing dates
  - DateTimeFormatter
- Dealing with different time zones
  - ZonedDateTime

‘’Dating’’ before Java 8

- java.util.Date class...
- java.util.Calendar class...
- java.text.DateFormat class...

- Joda-Time

LocalDate

- Immutable object
- Plain date without the time of day
- Does not carry any information about the time zone
- Instance created using the *of* static factory method
- Many methods to read its most commonly used values such as year, month, day of the week, and so on

```
LocalDate today = LocalDate.now();                    2016-03-15
LocalDate date1 = LocalDate.of(2016, 6, 5);           2016-06-05
LocalDate date2 = LocalDate.of(2010, Month.JANUARY, 1);  2010-01-01
```

LocalTime

- Instances of LocalTime are created using three overloaded static factory methods named *of*
- The first one accepts an hour and a minute and the second one also accepts a second. The third one - nanoseconds
- Just like the LocalDate class, the LocalTime class provides some getter methods to access its values

```
LocalTime time1 = LocalTime.of(23, 00);
LocalTime time2 = LocalTime.of(3, 10, 20);
LocalTime time3 = LocalTime.now();
```

LocalDate + LocalTime = LocalTimeDate

- Composite class pairs a LocalDate and a LocalTime
- Represents both a date and a time, without a time zone, and can be created either directly or by combining a date and time

```
LocalDateTime dt1 = LocalDateTime.of(2014, Month.MARCH, 18,13,45,20);
                        //2014-03-18T13:45:20
LocalDate date = LocalDate.of(2016, 6, 5);
LocalTime time = LocalTime.of(23, 00);
LocalDateTime dt2 = LocalDateTime.of(date, time); //2016-06-05T23:00
LocalDateTime dt3 = date.atTime(13, 45, 20); //2016-06-05T13:45:20
LocalDateTime dt4 = date.atTime(time); //2016-06-05T23:00
LocalDateTime dt5 = time.atDate(date); //2016-06-05T23:00
```

Instant

- A date and time for machines
- Represents the number of seconds passed since the Unix epoch time, set by convention to midnight of January 1, 1970 UTC.

```
Instant now = Instant.now();
Instant ofEpochSecond10 = Instant.ofEpochSecond(10);
// one milion nanosecond after 10' second
Instant ofEpochSecond100 = Instant.ofEpochSecond(10, 1_000_000);
```

Duration

- Duration class helps to measure duration between LocalTimes, LocalDateTimes, or two Instants
- LocalDateTime used by humans and Instant by machines are made for different purposes => do not mix them!
- Duration is used to represent an amount of time measured in seconds and eventually nanoseconds => do not pass a LocalDate to the between method

```
Temporal time1 = LocalTime.of(9, 00, 00);
Temporal time2 = LocalTime.of(17, 00, 00);
Temporal dateTime1 = LocalDateTime.of(2016, 1, 1, 0, 0, 0);
Temporal dateTime2 = LocalDateTime.of(2016, 1, 2, 0, 0, 0);
Temporal instant1 = Instant.ofEpochSecond(0);
Temporal instant2 = Instant.ofEpochSecond(3600);

Duration d1 = Duration.between(time1, time2); // 8 hours
Duration d2 = Duration.between(dateTime1, dateTime2); // 1440 minutes
Duration d3 = Duration.between(instant1, instant2); // 60 minutes
```

Period

- Period class models an amount of time in terms of years, months, and day

```
Period tenDays = Period.between(
   LocalDate.of(2014, 3, 8),
   LocalDate.of(2014, 3, 18)
);
int days = tenDays.getDays(); // 10 days
```

```
Period tenDays = Period.ofDays(10);
Period threeWeeks = Period.ofWeeks(3);
Period twoYearsSixMonthsOneDay = Period.of(2, 6, 1);
```

Manipulating

```
// 1st January 1989, 18:00:
Temporal dateTime = LocalDateTime.of(1989, 1, 20, 18, 00, 0);
Period twentySeven = Period.ofYears(27);
Period twentyDays = Period.ofDays(20);

// 31st January 2015, 18:00
Temporal result = dateTime.plus(twentySeven).minus(twentyDays);
```

Parsing & Formatting

- java.time.format.DateTimeFormatter
- Create a formatter is through its static factory methods and constants. The constants such as BASIC_ISO_DATE and ISO_LOCAL_DATE

```
LocalDate date = LocalDate.of(2016, 3, 15);
String basicFormat = date.format(DateTimeFormatter.BASIC_ISO_DATE); // 20160315
String isoFormat = date.format(DateTimeFormatter.ISO_LOCAL_DATE); // 2016-03-15
```

Creating a DateTimeFormatter from a pattern

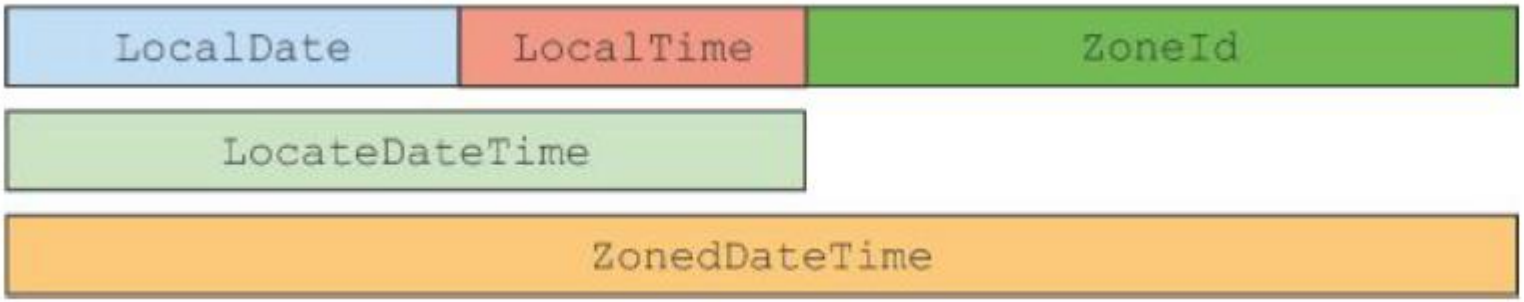
```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("MM/dd/yyyy");
LocalDate americanStyle = LocalDate.of(2016, 3, 15);
String formattedDate = americanStyle.format(formatter); // 03/15/2016
```

Creating a localized DateTimeFormatter

```
DateTimeFormatter germanFormatter = DateTimeFormatter.ofPattern(
                "d. MMMM yyyy", Locale.GERMAN);

LocalDate date = LocalDate.of(2016, 3, 15);
String germanDate = date.format(germanFormatter); //15. März 2016
```

Time Zones

- New java.time.ZoneId class is the replacement for the old java.util.TimeZone class
  - ZoneId warsawZone = ZoneId.of("Europe/Warsaw");
- Format “{area}/{city}” and the set of available locations is the one supplied by the IANA Time Zone Database (link)
- Translate old TimeZone object to a ZoneId by using the new method toZoneId:
  - ZoneId zoneId = TimeZone.getDefault().toZoneId();



Time Zones

```
ZoneId moscowZone = ZoneId.of("Europe/Moscow");

LocalDate date = LocalDate.of(2016, Month.MARCH, 15);
// 2016-03-15T00:00+03:00[Europe/Moscow]
ZonedDateTime zdt1 = date.atStartOfDay(moscowZone);

LocalDateTime dateTime = LocalDateTime.of(2016, Month.JANUARY, 1, 12, 00);
// 2016-01-01T12:00+03:00[Europe/Moscow]
ZonedDateTime zdt2 = dateTime.atZone(moscowZone);

Instant instant = Instant.now();
// 2016-03-14T22:55:34.280+03:00[Europe/Moscow]
ZonedDateTime zdt3 = instant.atZone(moscowZone);
```

Summary

- The old *java.util.Date* class and all other classes used to model date and time in Java have many inconsistencies and design flaws, including their mutability and some poorly chosen offsets, defaults, and naming.
- The date-time objects of the new Date and Time API are all immutable.
- This new API provides two different time representations to manage the different needs of humans and machines when operating on it.
- The results of date and time manipulations are always a new instance, leaving the original one unchanged.
- The formatter defines printing and parsing date-time objects in a specific format. These formatters can be created from a pattern or programmatically and they are all thread-safe.
- The time zone are represented relative to a specific region/location.