else {
array_push($gue...$post);
} ?>
<!-- Stop The Loop (but note the "else:" - see
<?php endwhile; else: ?>
<!-- The very first "if" tested to see if there were
display. This "else" part tells what do if there
no posts matched your criteria.</

Code.Hub

The first Hub for Developers

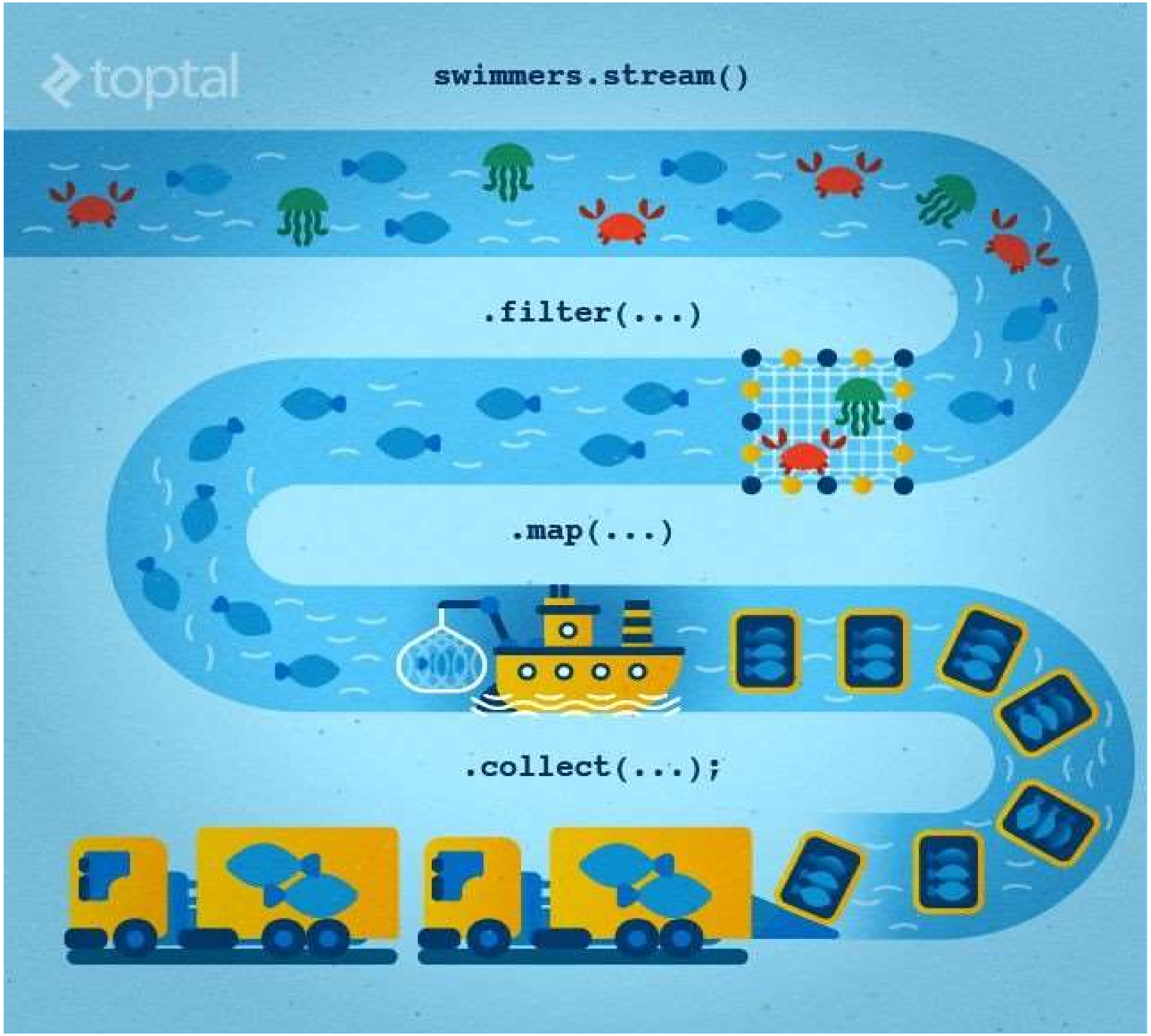
Java VII
Java 8 Features

Contents

- ✓ Streams API
- ✓ Lambda Expressions
- ✓ Functional Interface
- ✓ Lambdas & Streams
- ✓ Optionals
- ✓ Date & Time API

Java 8 Stream

Streams:



Streams

A stream is a sequence of elements.

- ✓ Streams are wrappers around a data source allowing us to operate with that data source and making bulk processing convenient and fast.
- ✓ A stream does not store data and, in that sense, is not a data structure. Instead, a stream carries values from a source through a pipeline
- ✓ It also never modifies the underlying data source.

Streams:
Stream Creation

From an existing array:

-
  ```
  private Person[] personsArray= {
      new Person("Spyros A", 34),
      new Person("Chris G", 28),
      new Person("Chris P", 36),
      new Person("Giannis V", 29)
  };
  Stream.of(personsArray);
  ```

From individual objects:

-
  ```
  Stream.of(personsArray[1], personsArray[0], personsArray[3]);
  ```

Streams:
Stream Creation

From an existing collection:

-
  ```
  private List<Person> persons = Arrays.asList(personsArray);
  persons.stream();
  ```

Note that **Java 8 added a new *.stream()* method to the *Collection* interface.**

Most common creation method

Streams:
Stream Creation

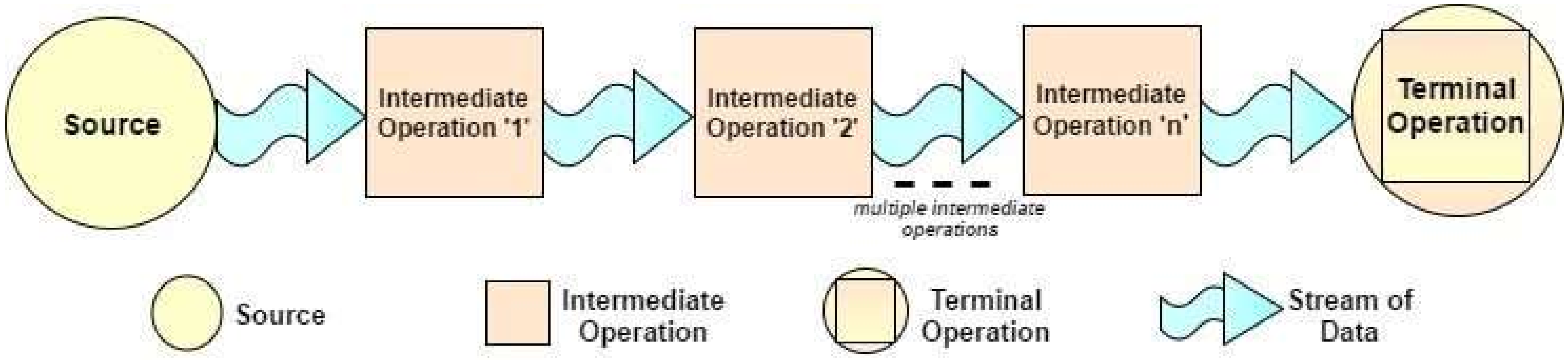
Using *Stream.builder()*:

- `Stream.Builder<Person> builder = Stream.builder();`

```
builder.accept(personsArray[0]);
builder.accept(personsArray[1]);
builder.accept(personsArray[2]);

Stream<Person> stream = builder.build();
```

NOT going to use it..

Streams: Aggregate Operations


Source
Intermediate Operation '1'
Intermediate Operation '2'
multiple intermediate operations
Intermediate Operation 'n'
Terminal Operation
Source
Intermediate Operation
Terminal Operation
Stream of Data

Streams: Aggregate Operations

Aggregate or Stream Operations are operations performed on a data structure as a whole, rather by iterating each element one by one.

✓ Before:

```
for (Person p : persons) {
    System.out.println(p.getName());
}
```

✓ After:

```
persons.stream()
       .filter(p -> p.getAge() > 30)
       .forEach(p -> System.out.println(p.getName());
```

Streams:
Aggregate Operations

Aggregate Operations are of two kinds

- Intermediate operations:

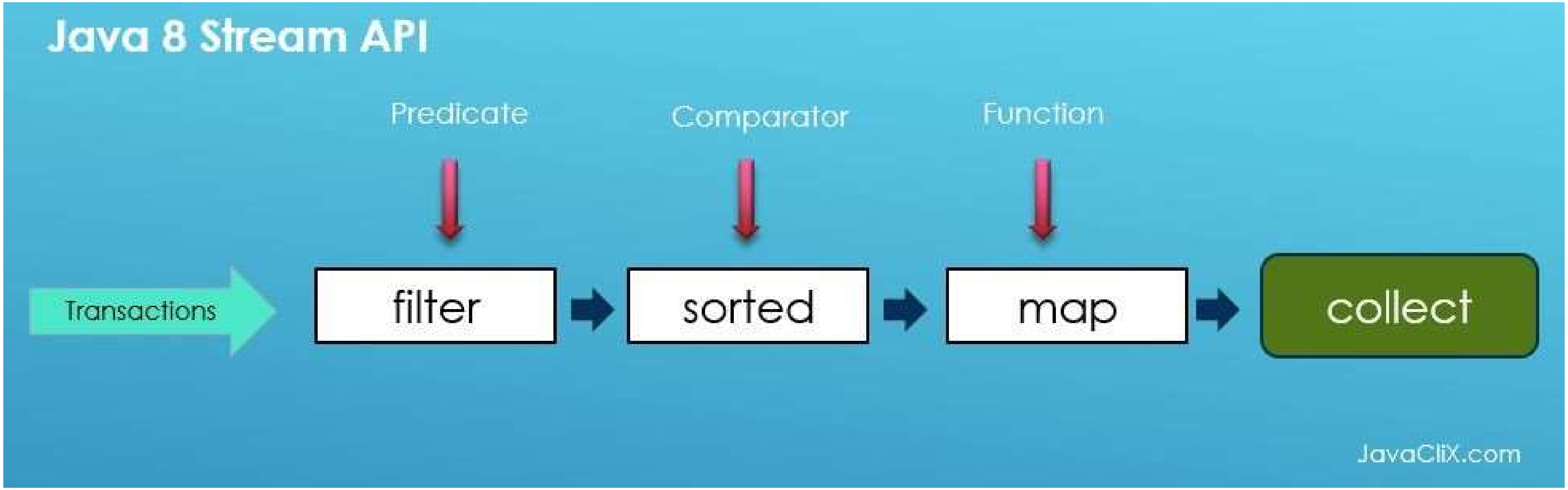
  An intermediate operation which produces a **new** stream

- Terminal operations:

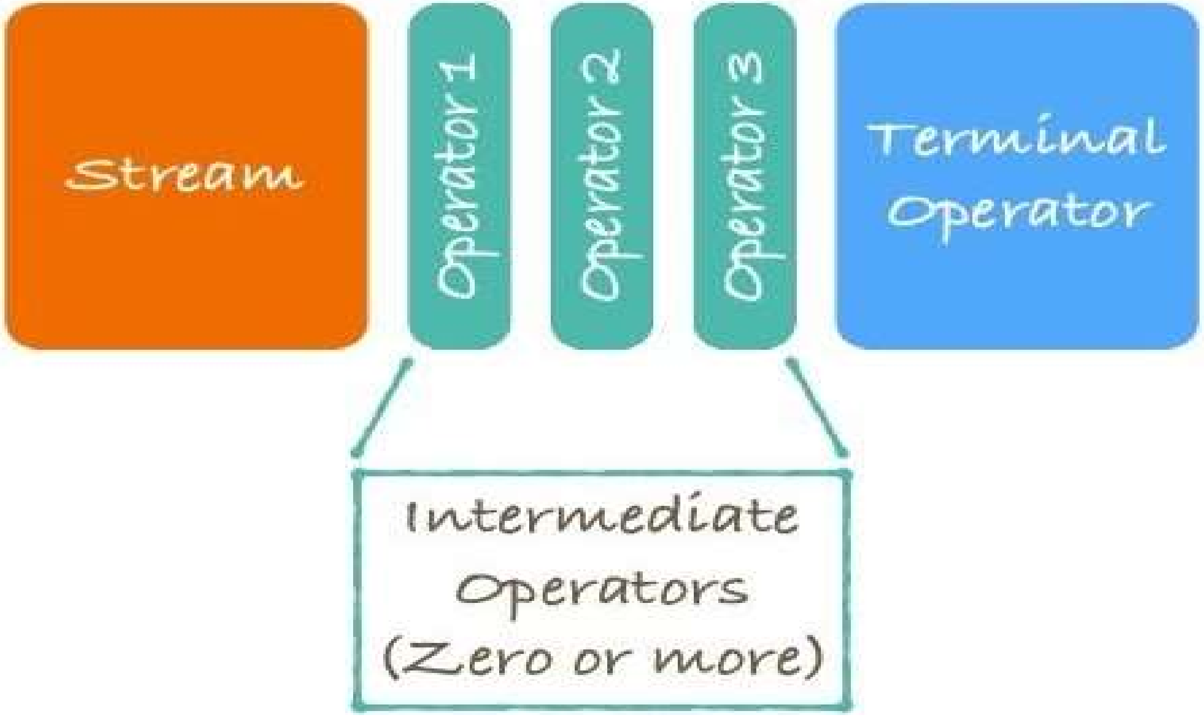
  A terminal operation, produces a non-stream result.

  - such as a primitive value (double, int, etc),
  - a collection (List, Set, etc),
  - or no value at all.

Streams: Aggregate Operations



Streams: Pipeline



Streams: Pipeline

A pipeline contains the following components:

- A source: This could be a collection, an array, a generator function, or an I/O channel.

  In this example, the source is the collection roster.

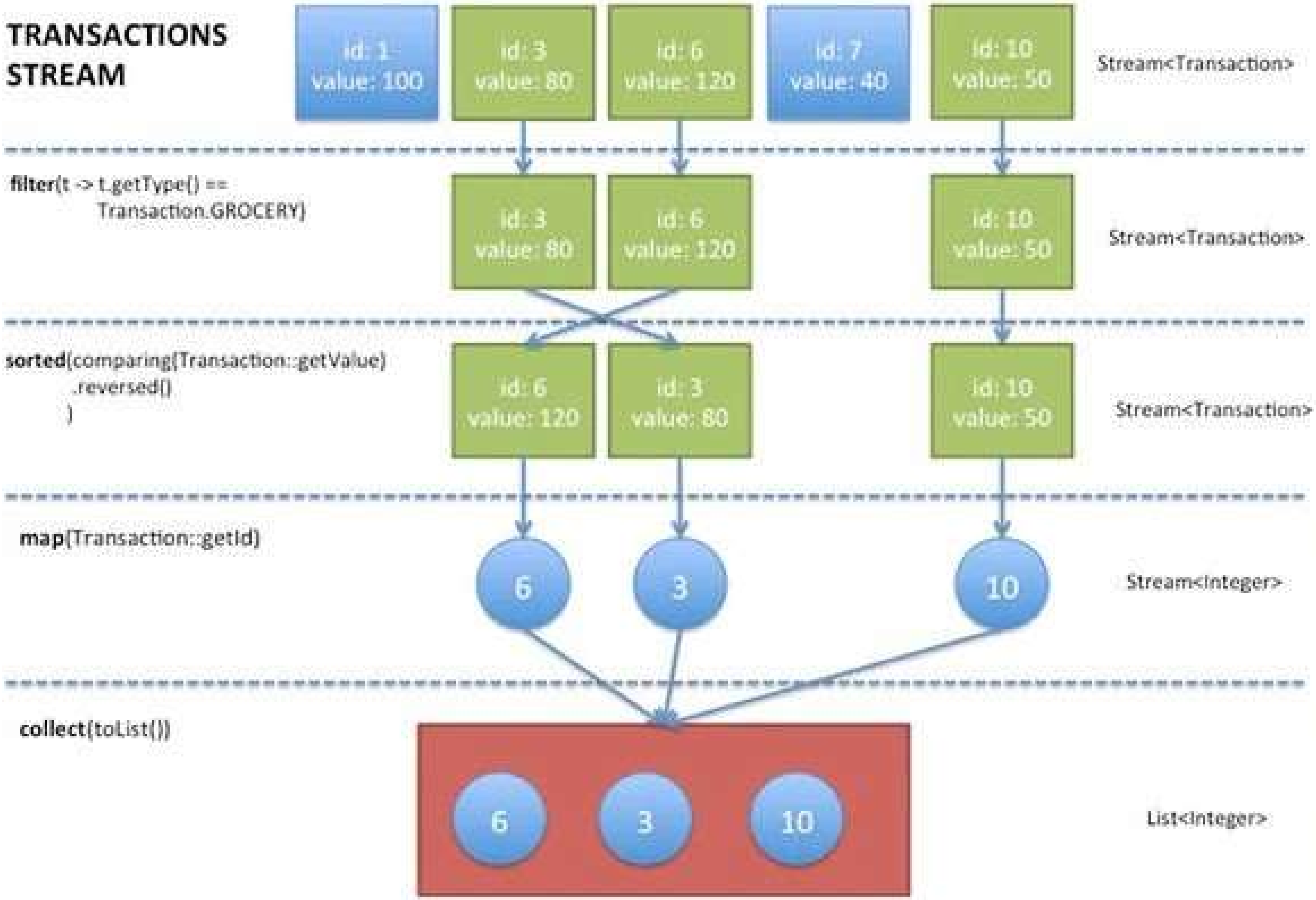
- Zero or **more** intermediate operations.

  An intermediate operation, such as `filter()`, produces a new stream.

- A terminal operation.

  A terminal operation, such as `forEach()`, produces no value at all.

Streams



TRANSACTIONS STREAM

id: 1 value: 100

id: 3 value: 80

id: 6 value: 120

id: 7 value: 40

id: 10 value: 50

Stream<Transaction>

filter(t -> t.getType() ==
Transaction.GROCERY)

id: 3 value: 80

id: 6 value: 120

id: 10 value: 50

Stream<Transaction>

sorted(comparing(Transaction::getValue)
.reversed()
)

id: 6 value: 120

id: 3 value: 80

id: 10 value: 50

Stream<Transaction>

map(Transaction::getId)

6

3

10

Stream<Integer>

collect(toList())

6

3

10

List<Integer>

Aggregate Operations VS Iterators

Aggregate operations, appear to be like iterators. However, they have several fundamental differences:

- They use internal iteration (*internal delegation*):

  Your application determines what collection it iterates, but the JDK determines **how** to iterate the collection.

- They process elements from a stream (*stream operations*):

  Aggregate operations process elements from a stream, not directly from a collection.

- They support behavior as parameters:

  You can specify lambda expressions as parameters for most aggregate operations.

  This enables you to customize the behavior of a particular aggregate operation.

```
(x,y) -> x > y ? x : y
```

```
(int a, int b) -> a+b
```

```
() -> {}
```

Lambda Expressions

```
(e1, e2) -> e1.age - e2.age
```

Lambda Expressions

Java is a **pure** object oriented programming language.

- Everything in Java is an Object with the exception of primitive types.
- You can't define **top level functions** (functions that don't belong to a class) in Java.
- You can't pass a function as an argument, or return a function from another function.

So, what's the alternative?

A bit of History lesson.. hold on!

Lambda Expressions

Before lambda expressions were introduced, developers used to use Anonymous class syntax for passing functionality to other methods or constructors.

Ex: to compare 2 elements of a list we need a class implementing the Comparator<T> interface.

This interface has one method compare(T o1, T o2) that when overridden provides a way for 2 objects to be compared.

```
class Person {
    private String name;
    private int age;
    // Constructor,
   //Getters, Setters

}
class PersonComparator implements
Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.getAge() - p2.getAge();
    }
```

Lambda Expressions

But it is too much of boilerplate code to create a class and instantiate an object just for using one method! So instead we use an anonymous class

```
List<Person> persons = Arrays.asList(new Person("Spyros A", 34),
        new Person("Chris G", 26));


// Sort persons based on their age by passing an anonymous Comparator.
persons.sort(new Comparator<Person>() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.getAge() - p2.getAge();
    }
});
```

Lambda Expressions

- ***You're getting the point right?***

- Since you can't pass functions directly as method arguments, You need to write all that boilerplate code all the time.

  Anonymous class syntax is more compact than defining a named class, instantiating it and then passing the instance as an argument.

  But still it's too much for classes with only one method!

Lambda Expressions

- ***Can we do better?***

- Is there a simpler way of passing a single functionality to other methods?

  Welcome to Lambda Expressions!

  History lesson over!

Lambda Expressions

- ***So what is Lambdas?***

A lambda expression is a block of code that gets passed around.

You can think of a lambda expression as an anonymous method.

It has parameters and a body just like full-fledged methods do,

but it doesn’t have a name like a real method.

In other words, a lambda expression

is like a method that you can pass as if it were a variable.

Lambda Expressions

Lambda expression allows you to pass functionality to other methods in a less verbose and more readable way, let’s rewrite the comparator compare method

```
persons.sort((Person p1, Person p2) -> {
        return p1.getAge() - p2.getAge();
});
```

If the method body consists of a single line, then you can omit the curly braces and the return:

```
persons.sort((Person p1, Person p2) -> p1.getAge() - p2.getAge());
```

Moreover, Since Java is aware about the types, you can omit the type declarations as well:

```
persons.sort((p1, p2) -> p1.getAge() - p2.getAge());
```

This is so concise, **readable**, and to the point.

Lambda Expressions

- ***Is that all? Are Lambdas just anonymous class replacements?***

- Lambdas are a lot more than just anonymous class replacements

  and they do not work automagically! Lambdas are based on **`@Functional Interfaces`**.

  Which are, in simple words, interfaces with only one method.

  So by passing the lambda expression, `() -> {},`

  you are passing the implementation of this one method!

Functional Interface

Let’s look at a basic functional interface example:

```
@FunctionalInterface
public interface Predicate<T> {
   boolean test(T t);
}
```

So Predicate is **any** lambda (block of code) that takes one object as input and returns a boolean.

- This is exactly how `.filter()` of stream works.
- It expects an (lambda) expression that can be evaluated as a boolean

Lambdas & Streams

```
public class Person {
    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    public int getAge() {...}
    //Constructor,Getters,Setters,toString,etc..
}
```

```
public enum Sex {
    MALE,
    FEMALE
}
```

Lambdas & Streams

```
persons.stream()
        .filter(
                p -> p.getGender() == Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25)
        .map(p -> p.getEmailAddress())
        .forEach(email -> System.out.println(email));
```

`filter()` is an intermediate aggregate operation performed on each element of the stream that returns an new stream containing only the elements for which the predicate returns true.

Lambdas & Streams

```
persons.stream()
       .filter(
               p -> p.getGender() == Person.Sex.MALE
               && p.getAge() >= 18
               && p.getAge() <= 25)
       .map(p -> p.getEmailAddress())
       .forEach(email -> System.out.println(email));
```

`map()` is an intermediate aggregate operation performed on each element of the stream that returns an new stream containing the result of the lambda expression evaluation

Lambdas & Streams

```
persons.stream()
       .filter(
               p -> p.getGender() == Person.Sex.MALE
               && p.getAge() >= 18
               && p.getAge() <= 25)
       .map(p -> p.getEmailAddress())
       .forEach(email -> System.out.println(email));
```

`forEach()` is an terminal aggregate operation, performed on each element of the stream, executing the lambda expression and that returns no value (void).

Functional Interface

```
@FunctionalInterface
public interface Consumer {
    void accept(T t);
}
```

So Consumer is **any** lambda (block of code) that takes one object as input and nothing.

It just executes the lambda.

- This is exactly how `.forEach()` of stream works.
- It expects an (lambda) expression and returns no value. That’s also why it’s terminal.

Lambdas & Streams

```
persons.stream()
        .filter(p -> p.getGender() == Person.Sex.MALE)
        .mapToInt(Person::getAge)
        .average()
        .getAsDouble();
```

What do you think this pipeline does?

Lambdas & Streams

```
double average =
         persons.stream()
                .filter(p -> p.getGender() == Person.Sex.MALE)
                .mapToInt(Person::getAge)
                .average()
                .getAsDouble();
```

What do you think this pipeline does?

Lambdas & Streams

```
double average =
          persons.stream()
                  .filter(p -> p.getGender() == Person.Sex.MALE)
                  .mapToInt(Person::getAge)
                  .average()
                  .getAsDouble();
```

`average()` is a terminal aggregate function that returns an the average value of all the elements of the stream

`average()` returns an `OptionalDouble` that is why `getAsDouble()` is needed.

Lambdas & Streams

```
persons.stream()
       .filter(p -> p.getGender() == Person.Sex.MALE)
       .map(Person::getName)
       .collect(Collectors.toList());
```

What do you think this pipeline does?

Lambdas & Streams

```
persons.stream()
        .filter(p -> p.getGender() == Person.Sex.MALE)
        .map(Person::getName)
        .collect(Collectors.toList());
```

What do you think this pipeline does?

`Person::getName` is a shorthand for `p -> p.getName()` lambda expression

This is called method reference.

Lambdas & Streams

```
List<String> names =
         persons.stream()
                .filter(p -> p.getGender() == Person.Sex.MALE)
                .map(Person::getName)
                .collect(Collectors.toList());
```

What do you think this pipeline does?

Lambdas & Streams

```
List<String> names =

         persons.stream()

                .filter(p -> p.getGender() == Person.Sex.MALE)

                .map(Person::getName)

                .collect(Collectors.toList());
```

`collect(Collectors.toList())` is a terminal aggregate function that returns a new List containing of all the elements of the stream.

Java 8
Optional

Optional

Optional is a class that wraps an object within and provides some specific functionality like:

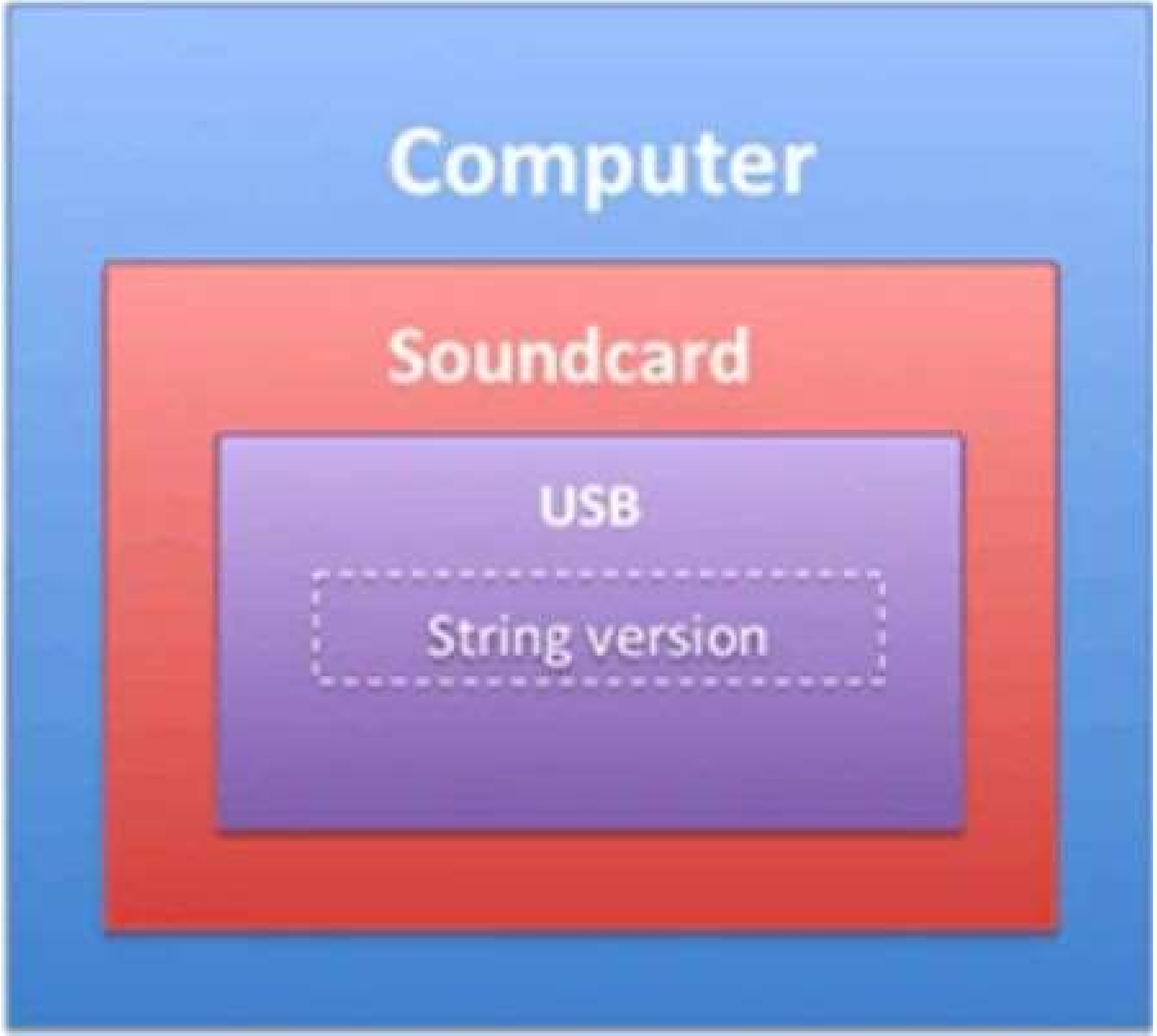
- isEmpty()
- isPresent()
- orElse(Object)
- orElseGet(Supplier)
- get()
- stream()
- map

And some more....

Optional

Optional are really useful to avoid null checks!

```
Optional<Dog> dog = Optional.of(new Dog(“Jack”));

dog.isEmpty()    - False

dog.isPresent() - True



Optional<String> s = Optional.empty();

s.orElse(“Some string”)
```


Computer
Soundcard
USB
String version

Old classic way

```
if (computer != null
       && computer.soundCard != null
       && computer.soundCard.usb != null
       && computer.soundCard.usb.version != null) {
   System.out.println(computer.soundCard.usb.version.toUpperCase());
} else {
   System.out.println("Version not found");
}
```

Optional way

```
String version = Optional.of(computer)
        .map(c -> c.getSoundCard())
        .map(s -> s.getUsb())
        .map(u -> u.getVersion())
        .map(v -> v.getVersion().toUpperCase())
        .orElse("No version found");
```

Which one is cleaner?

Exceptions

```
String version = Optional.of(computer)
        .map(c -> c.getSoundCard())
        .map(s -> s.getUsb())
        .map(u -> u.getVersion())
        .map(v -> v.getVersion())
        .orElseThrow(() -> new RuntimeException("No version found"));
```

Java 8 Date Time API

LocalDate, LocalDateTime, Instant

LocalDate

You can’t instantiate a LocalDate - Constructor is private!

```
LocalDate localDate = LocalDate.now();
localDate = localDate.minusDays(1);

localDate = localDate.minusMonths(1);

localDate = localDate.minusWeeks(1);

localDate = localDate.minusYears(1);



System.out.println(localDate.getDayOfMonth()); // [1 - 31]

System.out.println(localDate.getDayOfWeek()); // [MONDAY, TUESDAY, ..., SUNDAY]

System.out.println(localDate.getDayOfYear()); // [1 - 365]
```

LocalDate

```
LocalDate now = LocalDate.now();

LocalDate yesterday = LocalDate.now().minusDays(1);


System.out.println(now.isAfter(yesterday));

System.out.println(now.isBefore(yesterday));

System.out.println(now.isEqual(yesterday));


LocalDate christmas = LocalDate.of(2018, 12, 25);    // 2018-25-12

christmas = LocalDate.of(2018, Month.DECEMBER, 25);    // They are both equals

System.out.println(christmas);
```

LocalTime

```
LocalTime localTime = LocalTime.now();
localTime = localTime.plusHours(1);
localTime = localTime.minusMinutes(1);
localTime = localTime.minusSeconds(1);
localTime = localTime.minusNanos(1);


System.out.println(localTime.getHour());    // [0 - 24]
System.out.println(localTime.getMinute()); // [0 - 59]
System.out.println(localTime.getSecond()); // [0 - 59]
System.out.println(localTime.getNano());    // [0 - 10 ^ 9 - 1]
```

LocalTime

```
LocalTime now = LocalTime.now();

LocalTime yesterday = LocalTime.now().minusHours(24);


System.out.println(now.isAfter(yesterday));

System.out.println(now.isBefore(yesterday));

System.out.println(now.equals(yesterday));
```

LocalDateTime

```
LocalDateTime localDateTime = LocalDateTime.now();
localDateTime = localDateTime.minusDays(1);
localDateTime = localDateTime.minusMonths(1);
localDateTime = localDateTime.minusWeeks(1);
localDateTime = localDateTime.minusYears(1);
localDateTime = localDateTime.plusHours(2);
localDateTime = localDateTime.plusMinutes(3);
localDateTime = localDateTime.plusSeconds(15);
localDateTime = localDateTime.minusNanos(50);


System.out.println(localDateTime.getDayOfMonth()); // [1 - 31]
System.out.println(localDateTime.getDayOfWeek()); // [MONDAY, TUESDAY, ..., SUNDAY]
System.out.println(localDateTime.getDayOfYear()); // [1 - 365]
```

LocalDateTime

```
LocalDateTime now = LocalDateTime.now();

LocalDateTime yesterday = LocalDateTime.now().minusDays(1);


System.out.println(now.isAfter(yesterday    ));

System.out.println(now.isBefore(yesterday));

System.out.println(now.isEqual(yesterday));


LocalDateTime christmas = LocalDateTime.of(2018, 12, 25, 21, 59, 11);     // 2018-25-12T21:59:11

christmas = LocalDateTime.of(2018, Month.DECEMBER, 25, 21, 59, 11);     // They are both equals

System.out.println(christmas);
```

TemporalAdjusters

```
LocalDateTime now = LocalDateTime.now();


TemporalAdjuster adj = TemporalAdjusters.next(DayOfWeek.WEDNESDAY);

System.out.println(now .with(adj));


adj = TemporalAdjusters.firstDayOfMonth();

System.out.println(now .with(adj));


adj = TemporalAdjusters.firstDayOfNextMonth();

System.out.println(now .with(adj));
```

Instant

The instant class in the Java date time API represents a specific moment of the timeline.

The instant is defined as an offset since the origin (called an epoch).

The epoch is January 1rst 1970 - 00:00 Greenwhich mean time (GMT).

Why use Instant over LocalDateTime?

- ➢ LocalDateTime is like the clock on your wall. It represents the time of your **local** area.
- ➢ Instant is a moment of the timeline, counting nanoseconds.
- ➢ Instants are usually used as date types in the database. We need somehow different data systems in different countries to have the values. That wouldn't happen with a LocalDateTime….

Instant

```
        Instant localDateTime = Instant.now();

        localDateTime = localDateTime.minus(1, ChronoUnit.DAYS);

//        localDateTime = localDateTime.minus(1, ChronoUnit.MONTHS) Does not apply

//        localDateTime = localDateTime.minus(1, ChronoUnit.WEEKS);    Does not apply

//        localDateTime = localDateTime.minus(1, ChronoUnit.YEARS);    Does not apply

        localDateTime = localDateTime.plus(2, ChronoUnit.HOURS);
        localDateTime = localDateTime.plus(3, ChronoUnit.MINUTES);

        localDateTime = localDateTime.plusSeconds(15);

        localDateTime = localDateTime.minusNanos(50);



        System.out.println(localDateTime.getEpochSecond());
```

Zones

```
ZoneId greeceZone = ZoneId.of("Europe/Athens");

Instant now = Instant.now( );

System.out.println("Time in Greece is now: " + now.atZone(greeceZone));


ZoneId chicagoZone = ZoneId.of("America/Chicago");

now = Instant.now( );

System.out.println("Time in Chicago is now: " + now.atZone(chicagoZone));


Instant utcNow = Instant.now();

System.out.println("UTC time is now: " + utcNow);
```

Period

```java
LocalDate today = LocalDate.now();
LocalDate myBirthday = LocalDate.of(1992, Month.NOVEMBER, 25);

long daysFromMyBirthday = ChronoUnit.DAYS.between(myBirthday, today);
long monthsFromMyBirthday = ChronoUnit.MONTHS.between(myBirthday, today);
long yearsFromMyBirthday = ChronoUnit.YEARS.between(myBirthday, today);

System.out.println("Days from my birthday: " + daysFromMyBirthday);
System.out.println("Months from my birthday: " + monthsFromMyBirthday);
System.out.println("Years from my birthday: " + yearsFromMyBirthday);


System.out.println();

Period period = Period.between(myBirthday, today);
System.out.println("I am " + period.getYears() + " years, " +
    period.getMonths() + " months and " +
    period.getDays() + " days old.");
```

Formatter

```
try {
  String input = "25-12-2018";
  DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-M-yyyy");
  LocalDate date = LocalDate.parse(input, formatter);
  System.out.println(date);

  input = "25-February-2018";
  formatter = DateTimeFormatter.ofPattern("dd-MMMM-yyyy");
  date = LocalDate.parse(input, formatter);
  System.out.println(date);
} catch (DateTimeParseException e) {
  // Handle the exception accordingly....
}
```

Formatter

```
try {

String input = "5:10 PM";

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("h:mm a");

LocalTime time = LocalTime.parse(input, formatter);

System.out.println(time);



formatter = DateTimeFormatter.ofPattern("MMMM d, YYYY");

LocalDate localDate = LocalDate.now();

String format = localDate.format(formatter);

System.out.println(format);

} catch (DateTimeParseException e) {

  // Handle the exception accordingly....

}
```