Java 8 Lambda expressions
&&
Java streams

Common programming styles

- Imperative
  - uses statements that change a program's **state**
  - an imperative program consists of **commands** for the computer to execute
  - focuses on describing **how** a program operates
- Declarative
  - expresses the **logic** of a computation **without** describing its **control flow**
- **Functional** (subtype of declarative programming)
  - treats computation as the evaluation of mathematical functions and let **state** and data **unchanged**
  - Attempts to **minimize** or eliminate **side effects**
  - **Immutability** (transparent & clear functions)

What are lambdas and **why** use them?

Lambdas are wrapped behaviours, logics.

1. Older techniques can be written in a cleaner form
   - anonym **inner class** (as a functional interface implementation)
2. A basic tool for **functional programming** in java (new)
3. Significant part in java 8 **streams** (new)

Lambda syntax 1/2

- A comma-separated **list of formal parameters** enclosed in parentheses.
    - eg.: *(Person p, Boolean b)->...*
  - You can **omit the data type** of the parameters in a lambda expression if not confusing.
    - *(p)->.....*
  - In addition, you can **omit the parentheses** if there is only one parameter.
    - *p->....*

p -> p.getGender() == Person. *Gender.MALE* *&& p.getAge() >= 18* *&& p.getAge() <= 25*	*(Person p) -> p.getGender() == Person.* *Gender.MALE* *&& p.getAge() >= 18* *&& p.getAge() <= 25*

(These expressions are the same (on the left “p” represents an instance of the Person class.))

Lambda syntax 2/2

- A **body** follows the arrow token, ->
  which consists of a single expression or a statement block.
  - If you specify a single expression, then the Java runtime evaluates the expression and then returns its value.
  - Alternatively, you can use a return statement:

```
p -> {
    return p.getGender() == Person.Gender.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25;
}
```

1. Lambdas as cleaner form

Example: Ways to filter a collection:

1. If
2. Method without parameter
3. Parametrized method
4. Local filter class
5. Filter class by interface
6. **Anonym class by functional interface**
7. **Lambda expressions**

From Anonym class by functional interface to lambda expression

- functional interface == contains just one abstract method, but can contain one or more *default* or static method (useful for creating lambdas)
- In java 8 it can be annotated as @FunctionalInterfce

With anonym class :

```
 btn.setOnAction(new
EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent event) {
            System.out.println("Hello World!");
        }
    });
```

With lambda:

```
  btn.setOnAction(
        event -> System.out.println
("Hello World!")
       );
```

Default method (also java 8 feature)

- add new real **functionality** to the **interface** (not just unimplemented methods)
- **Diamond** inheritance still **not** in Java: have to specify which implementation to use in child from the availables

```
public interface TimeClient {
  LocalDateTime getLocalDateTime();
  static ZoneId getZoneId (String zoneString){
        try {
              return ZoneId.of(zoneString);
         } catch (DateTimeException e) {
              return ZoneId.systemDefault();
        }
   }
   default ZonedDateTime getZonedDateTime(String zoneString) {
      return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
   }}
```

2. Functional programming

- Family of the built-in functional interfaces in Java 8:
  - Predicate, BiPredicate,IntPredicate,LongPredicate,DoublePredicate
  - Consumer, BiConsumer,IntConsumer,LongConsumer,DoubleConsumer
  - Function, BiFunction,IntFunction,LongFunction,DoubleFunction
  - Supplier
- Notes:
  - **Int**Predicate,IntConsumer,...are the primitive versions (int, long, double stored, on autoboxing needed)
  - The **Bi**nary versions can take 2 parameters, not just one

Predicate (BiPredicate)

- Checks a condition and returns a boolean value as result
- Abstract method declaration:
  - boolean test(T t);
- Primitive versions: IntPredicate, LongPredicate, DoublePredicate

```
public class PredicateTest {
	public static void main(String []args) {
		Predicate<String> nullCheck = arg -> arg != null;
		Predicate<String> emptyCheck = arg -> arg.length() > 0;
		Predicate<String> nullAndEmptyCheck = nullCheck.and(emptyCheck);
		String helloStr = "hello";
		System.out.println(nullAndEmptyCheck.test(helloStr));
		String nullStr = null;
		System.out.println(nullAndEmptyCheck.test(nullStr));
	}
}
```

Consumer (BiConsumer)

- Operation that takes an argument, but returns nothing
- Abstract method declaration:
  - void accept(T t);
  - andThen() - chaining calls to Consumer object
- Primitive versions: Int-Long-DoubleConsumer, ObjIntConsumer(Tt, int value)..

```
Consumer<Person> printDatas =
        p->
        System.out.println(
        “This “ +p.getGender() +”person is “ +p.getAge() +”old”);
```

Function (BiFunction)

- Functions that take an argument and return a result
- Abstract method declaration:
  - R apply(T t);
  - andThen() -first apply the current Function then the passed argument
  - compose() -first apply the passed argument, then the current Function
    - Function<String, Integer> parseAndAbsInt = absInt.compose(parseInt);
  - identity() -just returns the passed argument without doing anything - for testing
- Primitive versions:Int-Long-DoubleFunction,ToIntFunction..,IntToLong..
- UnaryOperator interface extends the Function interface (Math::abs)
- BinaryOperator interface extends the BiFunction interface

Function<String, Integer> strLength = str -> str.length();

Supplier

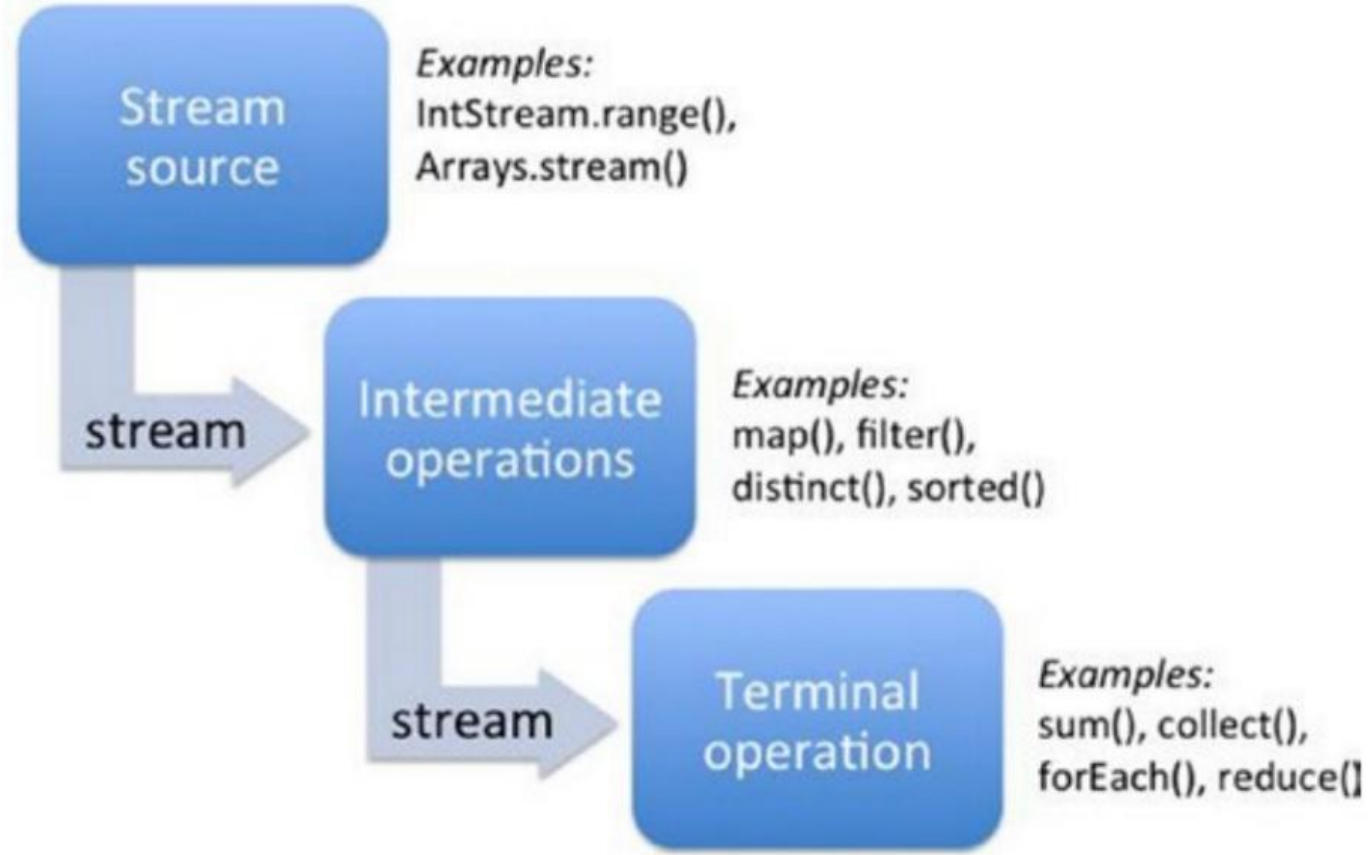
- Operation that returns a value to the caller
- Abstract method declaration:
  - T get();
- Primitive versions:Boolean-Int-Long-DoubleSupplier

```
Supplier<String> currentDateTime =
        () ->
        LocalDateTime.now().toString();
```

3. Java 8 streams

- Streams ~ data flow
- Pros:
  - In most cases more compact
  - In most cases has cleaner structure+ readability (focusing on and highlighting the logic)
  - In most cases easy to maintain
  - Easily scalable (parallelling easily due to immutability)
- Cons:
  - For small collections working with not parallel streams will cost more and will be slower than an old iteration
  - Difficult to test, exceptions?
- So for best results you have to think it over and measure, before refactor a code into java 8 style!
- Good to know about streams:
  - The source collection remains the original (due to immutability, so the result of the stream have to be collected and given to a variable)
  - Streams can be go along just once (no restart) else throw IllegalStateException
  - stream pipeline is processing the elements one by one

What can stream pipes build up from?



1. Sources

- By **stream methods**:
  - from primitive streams (eg.:IntStream,LongStream..)
    - range(int from, int toNotIncluded), rangeClosed - IntStream.range(1, 6)
    - iterate(1,i->i+1) - infinitive - IntStream.iterate(1, i -> i + 1)
  - of() - Stream.of(1, 2, 3, 4, 5) Stream.of(new Integer[]{1, 2, 3, 4, 5})
  - generate (Supplier<T> s)- Stream.generate(()->LocalDateTime.now())
  - builder() + add() - Stream.builder().add(1).add(2).add(3).add(4).add(5).build()
- From an **array** + stream()
  - Arrays.stream(new int[] {1, 2, 3, 4, 5})
- From a **Collection**
  - interface has been extended by
    - toStream()
    - toParalellStream()

1.Sources, advanced level:

- lines() in java.nio.file.Files class
  - Files.lines(Paths.get("./FileRead.java")).forEach(System.out::println);
- splitAsStream() in java.util.regex.Pattern class
  - Pattern.compile(" ").splitAsStream("java 8 streams").forEach(System.out::println);
- ints() in java.util.Random class
  - new Random().ints().limit(5).forEach(System.out::println);
- chars() in java.lang.String class
  - "hello".chars().sorted().forEach(ch -> System.out.printf("%c ", ch)); // prints e h l l o, works with IntStream - (there is no primitive char stream)

2. Intermediate operations

- Intermediate operations (or stream pipe-elements) executes the logic.
- The intermediate operation returns an another stream, which can be modified by an another operation
- Omittable from stream pipeline
- Common intermediate operations are the following:
  a. Stream<T> filter(Predicate<?super T> check)
  b. <R> Stream<R> map(Function<?super T,? extends R> transform)
  c. Stream<T> distinct()
  d. Stream<T> sorted() Stream<T> sorted(Comparator<?super T> compare)
  e. Stream<T> peek(Consumer<? super T> consume)
  f. Stream<T> limit(long size)

a. Filter

- Stream<T> filter(Predicate<?super T> check)
- Removes the elements for which the check predicate returns false.

```
IntStream.rangeClosed(0, 10)
            .filter(i -> (i % 2) == 0)
            .forEach(System.out::println);

//This will print: 0 2 4 6 8 10

//the following method also can be used in filter, or even as methodReference!
public static boolean isEven(int i)
{
      return (i % 2) == 0;
}
```

b. Map

- <R> Stream<R> map(Function<?super T,? extends R> transform)
- Applies the transform() function for each of the elements in the stream.

```
IntStream.rangeClosed(0, 10)
            .map(i -> i * i)
            .filter(i -> (i % 2) == 0)
            .forEach(System.out::println);

//This will print 0 4 16 36 64 100
```

c. Distinct

- Stream<T> distinct()
- Removes duplicate elements in the stream by equals() method

```
IntStream chars = "bookkeep"
                        .chars();

chars.distinct().forEach(ch -> System.out.printf("%c ", ch));

//This will print: b o k e p
```

d. Sorted

- Stream<T> sorted()
- Stream<T> sorted(Comparator<?super T> compare) (lamda can be used as Comparator)
- Sorts the elements in its natural order.

```
IntStream chars = "hello"
                  .chars()
                  .sorted(); // there is no execution yet!

chars.forEach(ch -> System.out.printf("%c ", ch));

 //This will print: e h l l o
```

e. Peek

- Stream<T> peek(Consumer<? super T> consume)
- Returns the same elements in the stream, but also executes the passed consume lambda expression on the elements.
- Use just for debugging

```
Stream.of(1, 2, 3, 4, 5)
        .peek(i -> System.out.printf("%d ", i))
        .map(i -> i * i)
        .peek(i -> System.out.printf("%d ", i))
        .count();

//This will print: 1 1 2 4 3 9 4 16 5 25
```

f. Limit

- Stream<T> limit(long size)
- Removes the elements if there are more elements than the given size in the stream.

```
IntStream.
        iterate(1, i -> i + 1).
        limit(5).
        forEach(System.out::println);

//This will print 1 2 3 4 5
```

Logic in an operation

- Can be written in:
  - Static method
  - Lambda (for small inline code, or in case of complex and not convenient)
  - *Method reference* (common usage, but not in case of complex and convenient context)
  - Functional interfaces (Predicate, Function,...)

Method references

- Use for reduce lamba-verbocity
  - If just routing parameters:
    - FROM: `strings.forEach(string -> System.out.println(string));`
    - TO: `strings.forEach(System.out::println);`
  - Can not used here:
    - `strings.forEach(string -> System.out.println(string.toUpperCase()));`
- Method references and Lambdas are similar in that they both require a target type that consist of a compatible functional interface.

Type (reference to a ...)	Syntax
static method	ClassName::staticMethodName
constructor	ClassName::new
instance method of an arbitary object of a particular type	ClassName::instanceMethodName
instance method of a particular object	containingObject::instanceMethodName

3. Terminal operations

- Terminal operations can return:
  - *Optional* object (for example Optional<Integer>)
  - Primitive value or an object
- Forces to produce the result.(without it, stream pipelines not evaluated)
- Common terminal operations:
  - toArray
  - Count
  - Min,Max
  - Reduce
  - Collect
  - ForEach
  - FindFirst, findAny
  - anyMatch, allMatch, noneMatch
  - Just in primitive streams
    - sum
    - average

Java 8 Optional

- No need for nullcheck and no more nullpointer exception
- Instead of null, a default value can be used by default
  - Eg.: in case of an Integer if the value is undefined, the optional can be Integer.ZERO by default
- BUT perform worse like manual nullcheck!

- void ifPresent(Consumer<? super T> consumer)
- boolean isPresent()
- Optional<T> of(T value)
- T get()- can throw NoSuchElementException
- Optional<T> ofNullable(T value)
- T orElse(T other)
- T orElseGet(Supplier<? extends T> other)
- <X extends Throwable> orElseThrow(Supplier<? extends X> exceptionSupplier)

```
String version = computer.getSoundcard().getUSB().
getVersion();

String version = "UNKNOWN";
if(computer != null){
        Soundcard soundcard = computer.getSoundcard();
        if(soundcard != null){...


Optional<Soundcard> soundcard = ...;
soundcard.ifPresent(System.out::println);
soundcard = soundcard.orElse(new Soundcard("base"));
soundcard = soundCard.orElseThrow
(IllegalStateException::new);
```

a. toArray

- <A> A[] toArray(IntFunction<A[]> generator);
- Returns an array containing the elements of this stream, using the provided generator function to allocate the returned array
- Not type safe (ArrayStoreException can be thrown)

```
Stream<String> stream = ...;
String[] stringArray = stream.toArray(size -> new String[size]);     //or
String[] stringArray = stream.toArray(String[]::new);
```

b. count

- long count()
- Returns the count of elements in this stream. This is a special case of a reduction

```
List<String> list = Arrays.asList("AA","AB","CC");

        Predicate<String> predicate = s-> s.startsWith("A");

        long l= list.stream().filter(predicate).count();

        System.out.println("Number of Matching Element:"+l);

//this will print : Number of Matching Element:2
```

c. min, max

- OptionalInt min() - for primitive typed Streams (eg.:IntStream in this case)
- Optional<T> min(Comparator<? super T> comparator)
- returns an Optional, describing the minimum/maximum of the elements, or an empty optional if the stream is empty.

```
IntStream myIntStream = IntStream.of(6,5,7,1, 2, 3, 3);
        OptionalInt minValue = myIntStream .min();
        if(minValue .isPresent())
        {
                System.out.println(minValue .getAsInt());
        }else{
                 System.out.println("no value");
        }
```

d. reduce

- T reduce(T identity, BinaryOperator<T> accumulator)
- Using the provided identity (as starting value) and an associative accumulation function, performs accumulation on the elements and returns the reduced value.

```
Integer totalAgeReduce = people
  .stream()
  .map(Person::getAge)
  .reduce(0, ((a, b) -> a + b));
```

```
Integer totalAge = people
    .stream()
    .map(Person::getAge)
    .reduce(0, Integer::sum)
```

e. collect

- <R,A> R collect(Collector<? super T,A,R> collector)
- <R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)
- Accumulate,so performs a mutable reduction operation on the elements of the stream (can use a Collector)
- Typesafe

```
List<String> asList = stringStream.collect(Collectors.toList());
```

```
Map<String, List<Person>> peopleByCity = personStream.collect(Collectors.groupingBy
(Person::getCity));
```

f. forEach

- void forEach(Consumer<? super T> action)
- Perform an action by a consumer with the corresponding type

```
List<String> strings = new ArrayList<>();
strings.stream().
            forEach(
            (string) -> {
            System.out.println("Content: " + string);
});
```

g. findFirst, findAny

- Optional<T> findFirst()
- Optional<T> findAny()
- Returns an Optional describing the first/any element of this stream, or an empty Optional if the stream is empty.(short circuiting)
- findAny is to allow for maximal performance in parallel operations

```
List<Integer> list = Arrays.asList(1, 10, 3, 7, 5);
int a = list.stream()
        .peek(num -> System.out.println("will filter " + num))
        .filter(x -> x > 5)
        .findFirst()
        .get();
System.out.println(a);
// this will print:   will filter 1    will filter 10   10
```

h. anyMatch, allMatch, noneMatch

- boolean anyMatch(Predicate<? super T> predicate)
- boolean allMatch(Predicate<? super T> predicate)
- boolean noneMatch(Predicate<? super T> predicate)
- Returns whether any/all/no elements of this stream match the provided predicate.(short circuiting)

```
Predicate<Employee> isNameStartsWithA = e -> e.id < 10 && e.name.startsWith
("A");
Predicate<Employee> isSalaryOver1000 = e -> e.sal < 10000;
List<Employee> list = Employee.getEmpList();
    System.out.println(list.stream().allMatch(isNameStartsWithA ));
    System.out.println(list.stream().anyMatch(isSalaryOver1000));
    System.out.println(list.stream().noneMatch(isNameStartsWithA));
(
```

i. Additional terminal operations in primitiveStreams: sum, average

- int sum() -Returns the sum of elements in this stream.
- OptionalDouble average() - Returns an OptionalDouble describing the arithmetic mean of elements of this stream, or an empty optional if this stream is empty. This is a special case of a reduction.

```
int sum = widgets.stream()
                .filter(w -> w.getColor() == RED)
                .mapToInt(w -> w.getWeight())
                .sum();
```

```
IntStream i = IntStream.of(6,5,7,1, 2, 3, 3);
        OptionalDouble value = i.average();
        if(value.isPresent())
        {
                System.out.println(
                value.getAsDouble());
        }else{
                System.out.println("no value");
        }
```

Links

- Programming paradigms: https://www.info.ucl.ac.be/~pvr/paradigmsDIAGRAMeng108.jpg
- Lambda step by step: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- Method reference: https://blog.idrsolutions.com/2015/02/java-8-method-references-explained-5-minutes/
- Java 8 and the performance: https://dzone.com/articles/applying-java-8-idioms-to-existing-code?edition=182488&utm_source=Daily%20Digest&utm_medium=email&utm_content=POS1&utm_campaign=dd%202016-06-12
- Default methods: https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html
- Functions: http://stackoverflow.com/questions/27872387/can-a-java-lambda-have-more-than-1-parameter
- Optional: http://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html
- Terminal operations: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#reduce-T-java.util.function.BinaryOperator-