Session 3 : Java Generics, Lambdas, Streams

sourcemind

Generics

Generics in Java enable “types” to be parameters in classes, interfaces, and methods.

GENERIC CLASSES

- The syntax for introducing a generic is to declare a formal type parameter in angle brackets.

```
public class Crate<T> {
    private T contents;
    public T emptyCrate() {
        return contents;
    }
    public void packCrate(T contents) {
        this.contents = contents;
    }
    }
```

The generic type **T** is available anywhere within the Crate class. When you instantiate the class, you tell the compiler what T should be for that particular instance.

NAMING CONVENTIONS FOR GENERICS

- **E** for an element
- **K** for a map key
- **V** for a map value
- **N** for a number
- **T** for a generic data type
- **S , U , V** , and so forth for multiple generic types

Example

```
Elephant elephant = new Elephant();

Crate<Elephant> crateForElephant = new Crate<>();

crateForElephant.packCrate(elephant);

Elephant inNewHome =
crateForElephant.emptyCrate();
```

```
Crate<Zebra> crateForZebra = new Crate<>();
```

Run-time type checking

```
class Shape { }
class Circle extends Shape { }
class Rectangle extends Shape { }
class Square extends Rectangle { }

List shapes = new ArrayList();
shapes.add(new Circle());
shapes.add(new Square());
shapes.add(new Circle());

Circle a = (Circle) shapes.get(0);
Square b = (Square) shapes.get(1);
Square c = (Square) shapes.get(2);
```

Compile-time type checking

```
List<Circle> shapes = new ArrayList<>();
shapes.add(new Circle());   // good
shapes.add(new Square());   // error
```

Raw types

- Prior to Java 5, generic types were not supported

```
// raw type
List list = getOldCollectionOfStrings();
// better
List<String> better = (List<String>) list;
```

- After introducing generics, it is allowed to use generic classes without type parameters: Raw types

```
Drawing draw = new Drawing(); // no type means Object

// Collections
List list = new List();
list.add("A");
list.add(1);
list.add(new Object());
```

Two generic parameters example

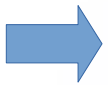
```
public class SizeLimitedCrate<T, U> {
    private T contents;
    private U sizeLimit;
    public SizeLimitedCrate(T contents, U sizeLimit) {
        this.contents = contents;
        this.sizeLimit = sizeLimit;
    }
    }
```

```
Elephant elephant = new Elephant();

Integer numPounds = 15_000;

SizeLimitedCrate<Elephant, Integer> c1 = new SizeLimiteCrate<>(elephant, numPounds);
```

TYPE ERASURE

Behind the scenes, the compiler replaces generics type with Object .

```
public class Crate<T> {
    private T contents;
    public T emptyCrate() {
        return contents;
    }
    public void packCrate(T contents) {
        this.contents = contents;
    }
    }
```



```
public class Crate {
    private Object contents;
    public Object emptyCrate() {
        return contents;
    }
    public void packCrate(Object contents)
    {
        this.contents = contents;
    }
    }
```

TYPE ERASURE

- This means there is only one class file. There aren't different copies for different parameterized types. (Some other languages work that way.)
- This process of removing the generics syntax from your code is referred to as **type erasure**. Type erasure allows your code to be compatible with older versions of Java that do not contain generics.
- The compiler adds the relevant casts for your code to work with this type of erased class. For example, you type the following:

- **Robot r = crate.emptyCrate();**
- The compiler turns it into the following:
- **Robot r = (Robot) crate.emptyCrate();**

GENERIC INTERFACES

```
public interface Shippable<T> {
    void ship(T t);
    }
```

```
class ShippableRobotCrate implements Shippable<Robot> {
    public void ship(Robot t) { }
    }
```

```
class ShippableAbstractCrate<U> implements Shippable<U> {
    public void ship(U t) { }
    }
```

```
class ShippableCrate implements Shippable {
    public void ship(Object t) { }
    }
```

WHAT YOU CAN'T DO WITH GENERIC TYPES

- **Calling a constructor**: Writing new T() is not allowed because at runtime it would be new Object() .
- **Creating an array of that generic type**: This one is the most annoying, but it makes sense because you'd be creating an array of Object values.
- **Calling instanceof** : This is not allowed because at runtime List<Integer> and List<String> look the same to Java thanks to type erasure.
- **Using a primitive type as a generic type parameter**: This isn't a big deal because you can use the wrapper class instead. If you want a type of int , just use Integer .
- **Creating a static variable as a generic type parameter**: This is not allowed because the type is linked to the instance of the class.

WHAT YOU CAN'T DO WITH GENERIC TYPES

- Cannot declare array of generics

```
Box<String>[] box = new Box<String>[3]; // Not allowed
```

- Reason: In Java, arrays are covariant while generics are invariant:

```
// Covariance in arrays: Compiles but throws runtime error
String[] array = new String[3];
Object[] objects = array;
objects[0] = 1;
String str = (String) objects[0];

// Invariance in generics: This would create issues in generics
Box<String>[] array = new Box<String>[5];
Box<Object> objects = array;
objects[0] = new Box<Object>(new Object()); // Against generics idea
```

- Recommendation: Use collections with generics

GENERIC METHODS

```
public class Handler {
    public static <T> void prepare(T t) {
        System.out.println("Preparing " + t);
    }
    public static <T> Crate<T> ship(T t) {
        System.out.println("Shipping " + t);
        return new Crate<T>();
    }
    }
```

OPTIONAL SYNTAX FOR INVOKING A GENERIC METHOD

You can call a generic method normally, and the compiler will try to figure out which one you want. Alternatively, you can specify the type explicitly to make it obvious what the type is.

```
Box.<String>ship("package");

Box.<String[]>ship(args);
```

BOUNDING GENERIC TYPES

- A **bounded parameter type** is a generic type that specifies a bound for the generic.
- A **wildcard generic type** is an unknown generic type represented with a question mark (**?**).

Types of bounds

Type of bound	Syntax	Example
Unbounded wildcard	`?`	`List<?> a = new ArrayList<String>();`
Wildcard with an upper bound	`? extends type`	`List<? extends Exception> a = new ArrayList<RuntimeException>();`
Wildcard with a lower bound	`? super type`	`List<? super Exception> a = new ArrayList<Object>();`

Unbounded Wildcards

```
public static void printList(List<?> list) {
    for (Object x: list)
        System.out.println(x);
    }
    public static void main(String[] args) {
        List<String> keywords = new ArrayList<>();
        keywords.add("java");
        printList(keywords);
    }
```

Upper-Bounded Wildcards

```
ArrayList<Number> list = new ArrayList<Integer>(); // DOES NOT COMPILE
```

```
List<? extends Number> list = new ArrayList<Integer>();
```

The **upper-bounded** wildcard says that any class that **extends Number or Number** itself can be used as the formal parameter type.

Lower-Bounded Wildcards

```
public static void addSound(List<? super String> list) {
    list.add("quack");
    }
```

```
List<String> strings = new ArrayList<String>();
strings.add("tweet");
List<Object> objects = new ArrayList<Object>(strings);
addSound(strings);
addSound(objects);
```

With a lower bound, we are telling Java that the list will be a list of String objects or a list of some objects that are a superclass of String .

PECS: Producer Extends Consumer Super

```
// Shape > Rectangle > Square
// When reading from a list (the list is a producer) use extends
// You guarantee that you are working with at least a Rectangle list
public void produce(List<? extends Rectangle> data) {
    Rectangle a = data.get(0);
    Shape b = data.get(1);
    data.add(new Rectangle());  // not allowed
}
// When adding to a list (the list is a consumer) use super
// You know then that the list can accept at least a Rectangle
public void consume(List<? super Rectangle> data) {
    data.add(new Rectangle());
    data.add(new Square());
    data.add(new Shape());
    Rectangle a = data.get(0);  // not allowed
}
```

- Use the <? extends T> wildcard if you need to retrieve object of type T from a collection.
- Use the <? super T> wildcard if you need to put objects of type T in a collection.
- If you need to satisfy both things, don’t use any wildcard.

Comparable<T> and Comparator<T>

- A sorted drawing object contains Shape objects that know how to compare to themselves.
- It supports any comparable shape class

```
class SortedDrawing<T extends Comparable<Shape>> {
    public void addShape() {
    }
    public void draw() {
    }
}
```

- Still, strongly types. Cannot mix comparable square and rectangle types

RectList<T extends Rectangle>

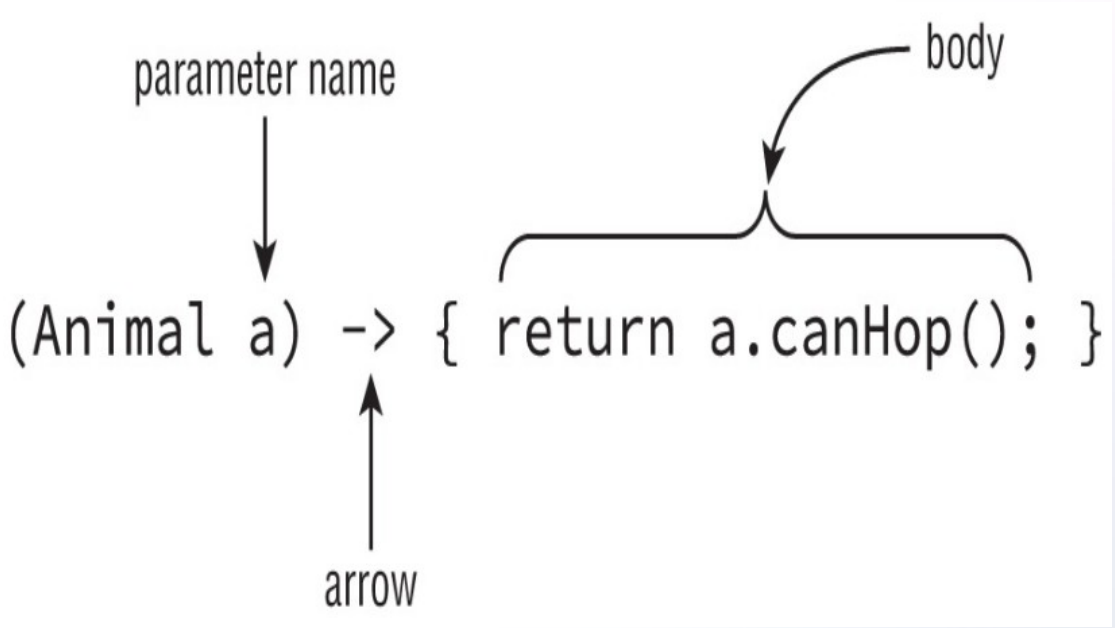
- Which statements are correct?

```
class RectList<T extends Rectangle> extends ArrayList<T> {
}

RectList<Rectangle> l1 = new RectList<>();
l1.add(new Rectangle());
l1.add(new Square());
l1.add(new Shape());

RectList<Square> l2 = new RectList<>();
l2.add(new Rectangle());
l2.add(new Square());
l2.add(new Shape());

RectList<Shape> l3 = new RectList<Shape>();
l3.add(new Rectangle());
l3.add(new Square());
l3.add(new Shape());
```

Lambdas and Functional Interfaces

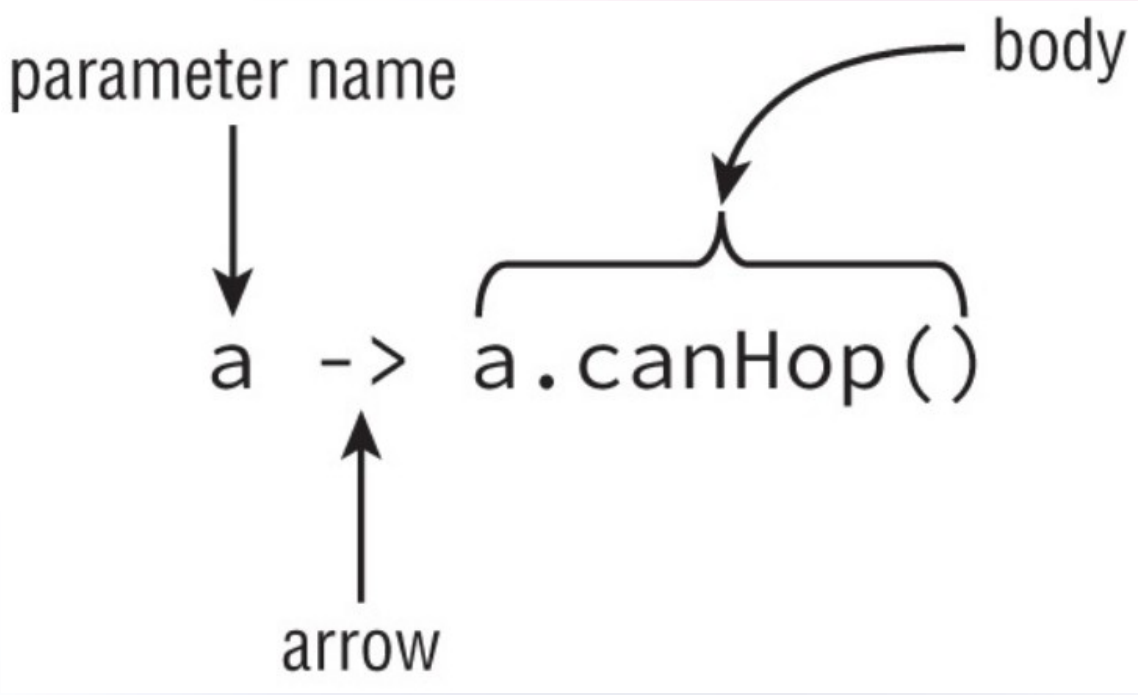
Lambdas

- A **lambda expression** is a block of code that gets passed around. You can think of a lambda expression as an unnamed method.



Optional parts

- The parentheses can be omitted only if there is a single parameter and its type is not explicitly stated
- We can omit braces when we have only a single statement
- Java doesn’t require you to type return or use a semicolon when no braces are used. This special shortcut doesn’t work when we have two or more statements



Valid lambdas

Lambda	# parameters
`() -> true`	0
`a -> a.startsWith("test")`	1
`(String a) -> a.startsWith("test")`	1
`(a, b) -> a.startsWith("test")`	2
`(String a, String b) -> a.startsWith("test")`	2

Invalid lambdas

Invalid lambda	Reason
`a, b -> a.startsWith("test")`	Missing parentheses
`a -> { a.startsWith("test"); }`	Missing `return`
`a -> { return a.startsWith("test") }`	Missing semicolon

Introducing Functional Interfaces

- Lambdas work with **interfaces that have only one abstract method**. These are called **functional interfaces**.
- **@FunctionalInterface** : marks an interface as functional interface

```
public interface CheckTrait {
    boolean test(Animal a);
    }
```

Lambdas and Functional interface

```
public class Animal {
    private String species;
    private boolean canHop;
    private boolean canSwim;

    public Animal(String speciesName, boolean
    hopper, boolean swimmer){
        species = speciesName;
        canHop = hopper;
        canSwim = swimmer;
    }
    public boolean canHop() { return canHop; }
    public boolean canSwim() { return canSwim; }
    public String toString() { return species; }
    }
```

```
public interface CheckTrait {
    boolean test(Animal a);
    }
```

```
public class CheckIfHopper implements
CheckTrait {
    public boolean test(Animal a) {
        return a.canHop();
    }
    }
```

Lambdas and Functional interface

```
import java.util.*;
public class TraditionalSearch {
    public static void main(String[] args) {
        // list of animals
        List<Animal> animals = new ArrayList<Animal>();
        animals.add(new Animal("fish", false, true));
        animals.add(new Animal("kangaroo", true, false));
        animals.add(new Animal("rabbit", true, false));
        animals.add(new Animal("turtle", false, true));
        // pass class that does check
        print(animals, new CheckIfHopper());
    }
    private static void print(List<Animal> animals, CheckTrait checker) {
        for (Animal animal : animals) {
        // the general check
        if (checker.test(animal))
            System.out.print(animal + " ");
        }
    System.out.println();
    }
    }
```

- Now what happens if we want to print the Animal s that swim? Sigh. We need to write another class, **CheckIfSwims** .

Lambdas and Functional interface

```
print(animals, new CheckIfHopper());
```

```
print(animals, a -> a.canHop());
```

- How about Animals that cannot swim?

```
print(animals, a -> ! a.canSwim());
```

Built-in functional interfaces

Predicate , Consumer , Supplier , Function, Comparator, etc

PREDICATE

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

CONSUMER

```
public interface Consumer<T> {
    void accept(T t);
    }
```

```
Consumer<String> consumer = x -> System.out.println(x);
```

SUPPLIER

```
public interface Supplier<T> {
    T get();
    }
```

```
Supplier<Integer> number = () -> 42;
Supplier<Integer> random = () ->new Random().nextInt();
```

COMPARATOR

```
public interface Comparator<T> {
    int compare(T o1, T o2);
    }
```

```
Comparator<Integer> ints = (i1, i2) -> i1 - i2;
```

FUNCTION

```
public interface Function<T, R> {
    R apply(T);
    }
```

Returning functional interfaces

- A method can have an interface as a return type, which can be constructed as a lambda
- Example: java.util.Comparator#comparingInt
- Creating a predicate

```
public Predicate<String> getCondition(int size) {
    return String str -> str.length() > size;
}

Predicate<String> condition1 = getCondition(3);
Predicate<String> condition2 = getCondition(10);

boolean test1 = condition1.test("Hello World");
boolean test2 = condition2.test("Hello World");
```

Closures in Java

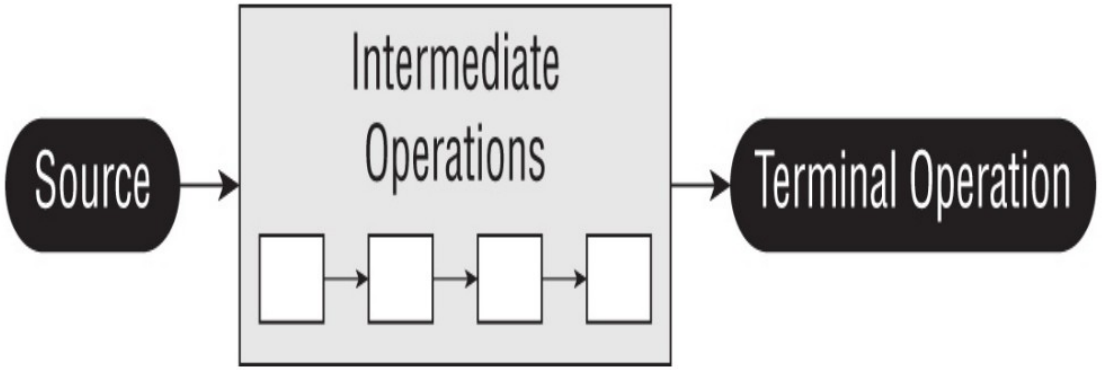
```
interface IntSupplier extends Supplier<Integer> {
}

IntSupplier[] amazing = new IntSupplier[10];
for (int i = 0; i < 10; i++) {
    int value = i;
    amazing[i] = () -> value;
}

System.out.println(i);
System.out.println(value);

for (int i = 0; i < 10; i++) {
    System.out.println(amazing[i].get());
}
```

Streams

Streams

- A **stream** in Java is a sequence of data.
- A **stream pipeline** consists of the operations that run on a stream to produce a result

Stream pipeline



- **Source**: Where the stream comes from
- **Intermediate operations**: Transforms the stream into another one. There can be as few or as many intermediate operations as you'd like. Since streams use lazy evaluation, the intermediate operations do not run until the terminal operation runs.
- **Terminal operation**: Actually produces a result. Since streams can be used only once, the stream is no longer valid after a terminal operation completes.

CREATING STREAM SOURCES

In Java, the streams we have been talking about are represented by the **Stream<T>** interface, defined in the **java.util.stream** package.

Creating Finite Streams

```
Stream<String> empty = Stream.empty(); // count = 0

Stream<Integer> singleElement = Stream.of(1); // count = 1

Stream<Integer> fromArray = Stream.of(1, 2, 3); // count = 3
```

Java also provides a convenient way of converting a Collection to a stream.

```
var list = List.of("a", "b", "c");
Stream<String> fromList = list.stream();
```

Creating Infinite Streams

```
Stream<Double> randoms = Stream.generate(Math::random);

Stream<Integer> oddNumbers = Stream.iterate(1, n -> n + 2);
```

TERMINAL OPERATIONS

- You can perform a terminal operation without any intermediate operations but not the other way around.
- **Reductions** are a special type of terminal operation where all of the contents of the stream are combined into a single primitive or Object .

Method	What happens for infinite streams	Return value	Reduction
`count()`	Does not terminate	`long`	Yes
`min()` `max()`	Does not terminate	`Optional<T>`	Yes
`findAny()` `findFirst()`	Terminates	`Optional<T>`	No
`allMatch()` `anyMatch()` `noneMatch()`	Sometimes terminates	`boolean`	No
`forEach()`	Does not terminate	`void`	No
`reduce()`	Does not terminate	Varies	Yes
`collect()`	Does not terminate	Varies	Yes

Examples : count(), min(), max()

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
System.out.println(s.count()); // 3
```

```
Optional<T> min(Comparator<? super T> comparator)
Optional<T> max(Comparator<? super T> comparator)
```

```
Stream<String> s = Stream.of("monkey", "ape", "bonobo");
Optional<String> min = s.min((s1, s2) -> s1.length()-s2.length());
min.ifPresent(System.out::println); // ape
```

Examples : reduce ()

```
T reduce(T identity, BinaryOperator<T> accumulator)

Optional<T> reduce(BinaryOperator<T> accumulator)

<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)
```

```
Stream<String> stream = Stream.of("w", "o", "l", "f");

String word = stream.reduce("", (s, c) -> s + c);

System.out.println(word); // wolf
```

Examples : collect()

```
<R> R collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R>  combiner)
<R,A> R collect(Collector<? super T, A,R> collector)
```

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
StringBuilder word = stream.collect(
StringBuilder::new,
StringBuilder::append,
StringBuilder::append)
System.out.println(word); // wolf
```

Examples : collect()

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
TreeSet<String> set = stream.collect(
TreeSet::new,
TreeSet::add,
TreeSet::addAll);
System.out.println(set); // [f, l, o, w]
```

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
TreeSet<String> set =
stream.collect(Collectors.toCollection(TreeSet::new));System.out.println(set); // [f, l, o, w]
```

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
Set<String> set = stream.collect(Collectors.toSet());
System.out.println(set); // [f, w, l, o]
```

INTERMEDIATE OPERATIONS

- Unlike a terminal operation, an intermediate operation produces a stream as its result. An intermediate operation can also deal with an infinite stream simply by returning another infinite stream.

filter(), distinct(), limit(), skip(), map(), flatMap(), sorted(), peek()

Examples : filter()

```
Stream<T> filter(Predicate<? super T> predicate)
```

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
s.filter(x -> x.startsWith("m"))
.forEach(System.out::print); // monkey
```

Examples : distinct()

```
Stream<T> distinct()
```

```
Stream<String> s = Stream.of("duck", "duck", "duck", "goose");
s.distinct()
.forEach(System.out::print); // duckgoose
```

Examples : limit() and skip()

```
Stream<T> limit(long maxSize)
Stream<T> skip(long n)
```

```
Stream<Integer> s = Stream.iterate(1, n -> n + 1);
s.skip(5)
.limit(2)
.forEach(System.out::print); // 67
```

Examples : map()

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
s.map(String::length)
.forEach(System.out::print); // 676
```

Examples : peek()

```
Stream<T> peek(Consumer<? super T> action)
```

```
var stream = Stream.of("black bear", "brown bear", "grizzly");
long count = stream.filter(s -> s.startsWith("g"))
.peek(System.out::println).count(); // grizzly
System.out.println(count); // 1
```

PRIMITIVE STREAMS

IntStream : Used for the primitive types **int , short , byte , and char**

LongStream : Used for the primitive type **long**

DoubleStream : Used for the primitive types **double and float**

```
IntStream intStream = IntStream.of(1, 2, 3);
OptionalDouble avg = intStream.average();
System.out.println(avg.getAsDouble()); // 2.0
```

Primitive stream methods

Method	Primitive stream	Description
OptionalDouble average()	IntStream, LongStream, DoubleStream	The arithmetic mean of the elements
Stream<T> boxed()	IntStream, LongStream, DoubleStream	A Stream<T> where T is the wrapper class associated with the primitive value
OptionalInt max(), min()	IntStream	The maximum element of the stream or The minimum element of the stream
OptionalLong max(), min()	LongStream	
OptionalDouble max(), min()	DoubleStream	

Primitive stream methods

IntStream range(int a, int b)	IntStream	Returns a primitive stream from a (inclusive) to b (exclusive)
LongStream range(long a, long b)	LongStream	
IntStream rangeClosed(int a, int b)	IntStream	Returns a primitive stream from a (inclusive) to b (inclusive)
LongStream rangeClosed(long a, long b)	LongStream	

Primitive stream methods

int sum()	IntStream	Returns the sum of the elements in the stream
long sum()	LongStream	
double sum()	DoubleStream	
IntSummaryStatistics summaryStatistics()	IntStream	Returns an object containing numerous stream statistics such as the average, min, max, etc.
LongSummaryStatistics summaryStatistics()	LongStream	
DoubleSummaryStatistics summaryStatistics()	DoubleStream	

Examples

```
DoubleStream oneValue = DoubleStream.of(3.14);
oneValue.forEach(System.out::println);

DoubleStream varargs = DoubleStream.of(1.0, 1.1, 1.2);
varargs.forEach(System.out::println);
```

```
var random = DoubleStream.generate(Math::random);
var fractions = DoubleStream.iterate(.5, d -> d / 2);
random.limit(3).forEach(System.out::println);
fractions.limit(3).forEach(System.out::println);
```

stream() and parallelStream()

Run the program using: - stream() - parallelStream()

```
public static void main(String[] args) {
    long N = 10_000_000;
    List<Long> listOfNumbers = new ArrayList<>();
    for (long i = 0; i < N; i++) listOfNumbers.add(i);

    long totalTime = 0;
    int times = 100;

    for (int i = 0; i < times; i++) {
        long now = System.currentTimeMillis();
        listOfNumbers.stream().reduce(Long::sum).get(); // TODO use parallelStream()
        totalTime += System.currentTimeMillis() - now;
    }

    System.out.println("Average runtime: " + (totalTime / (double) times) + " ms");
}
```

Homework 3

Go back to your solution of homework 1:

Read a large text line by line and count the number of lines and words.

Use streams to write as consice code as you can. Do not modify your old solution. Instead, add a new class that contains your new code and publish it as a new Maven version (core).

Update the dependency of service to use the new version. You should not change any source code in the service module, just update the dependency version to the new published version (e.g. 1.1.0).

Use the file provided in the repository.

Compare your new solution with the solution during homework 1.

Sample output (using Unix wc tool) to verify the correctness of your program:

```
cat input-file.txt | wc -wl
     16     704
```