Functional Java

Advanced Java I. Functional, Asynchronous, Reactive Java

Module 1

think.
create.
accelerate.

luxoft | training
A DXC Technology Company

Java Advanced I: Functional, Asynchronous, Reactive Java

Java Advanced: Functional, Asynchronous, Reactive Java

Subjects included in the course (30 hours):

- Functional Java: functional interfaces, streams
- Executor framework and Fork Join pool
- NIO – non-blocking input/output
- Asynchronous Java (Completable Future)
- Reactive Streams (Java 9)
- RxJava 2
- R2DBC (reactive JDBC replacement)
- Spring WebFlux (Reactor)
- Reactive Spring Data JPA

Functional interfaces
Method references

A Lambda Expression

Let’s use an anonymous class

```
FileFilter fileFilter = new FileFilter() {
    @Override
    public boolean accept(File file) {
        return file.getName().endsWith(".java");
    }
};
```

We take the parameters and return:

```
FileFilter filter = (File file) -> file.getName().endsWith(".java");
```

This is a lambda expression.

Several ways of writing a Lambda Expression

The simplest way:

```
FileFilter filter = (File file) -> file.getName().endsWith(".java");
```

If you have more than one line of code:

```
Runnable r = () -> {
   for (int i = 0; i < 5; i++) {
      System.out.println("Hello world!");
   }
};
```

If you have more than one argument:

```
Comparator<String> c =
      (String s1, String s2) -> Integer.compare(s1.length(), s2.length());
```

What is the type of a lambda expression?

=> *Functional interface*

What is a functional interface?

A functional interface is an interface with **only one *abstract* method**

(methods from class **`Object`** do not count)

```
public interface Runnable {
    run();
}

public interface Comparator<T> {
    int compareTo(T t1, T t2);
}

public interface FileFilter {

    boolean accept(File pathname);

}
```

Functional interfaces

A functional interface can be annotated:

```
@FunctionalInterface
public interface MyFunctionalInterface {
    someMethod();
     /**
      * Some more documentation
     */
    equals(Object o);
};
```

The annotation is here just for convenience, the compiler can tell me whether the interface is functional or not

Functional interfaces

Functional interface	Descriptor	Method name
Predicate<T>	T -> boolean	test()
BiPredicate<T, U>	(T, U) -> boolean	test()
Consumer<T>	T -> void	accept()
BiConsumer<T, U>	(T, U) -> void	accept()
Supplier<T>	() -> T	get()
Function<T, R>	T -> R	apply()
BiFunction<T, U, R>	(T, U) -> R	apply()
UnaryOperator<T>	T -> T	identity()
BinaryOperator<T>	(T, T) -> T	apply()

Examples:

```
Predicate<Integer> isAdult = age -> age >= 18;
isAdult.test(10);
Consumer<String> printer = p -> System.out.println("Printed: "+p);
printer.accept("hi");
Supplier<String> sayHi = () -> "hi";
sayHi.get(); // hi
```

Package java.util.function

Categories:

Supplier

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

Consumer / BiConsumer

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

```java
@FunctionalInterface
public interface BiConsumer<T, U> {
    void accept(T t, U u);
}
```

Predicate

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

BiPredicate

```java
@FunctionalInterface
public interface BiPredicate<T, U> {
    boolean test(T t, U u);
}
```

Function

```java
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

Unary operator

```java
@FunctionalInterface
public interface UnaryOperator<T>
extends Function<T, T> {
}
```

Omitting parameter

```java
Comparator<String> c =
(String s1, String s2) -> Integer.compare(s1.length(), s2.length());
```

Becomes:

```java
Comparator<String> c =
(s1, s2) -> Integer.compare(s1.length(), s2.length());
```

Method references

```
Function<String, String> f = s -> s.toLowerCase();
```

Can be written like that:

```
Function<String , String> f = String::toLowerCase;            f.apply("Hi") // hi
```

This lambda expression:

```
Consumer<String> c = s -> System.out.println(s);
```

Can be written like that:

```
Consumer<String> c = System.out::println;
```

This lambda expression:

```
Comparator<Integer> c = (i1, i2) -> Integer.compare(i1, i2);
```

Can be written like that:

```
Comparator<Integer> c = Integer::compare;
```

Example:
LambdaTest

Method references

```
Arrays.sort(rosterAsArray,
    (Person a, Person b) -> {
        return a.getBirthday()
               .compareTo(b.getBirthday());
    }
);
```

However, this method to compare the birth dates of two Person instances already exists as Person.compareByAge. You can invoke this method instead in the body of the lambda expression:

```
Arrays.sort(rosterAsArray,
       (a, b) -> Person.compareByAge(a, b));
```

Because this lambda expression invokes an existing method, you can use a method reference instead of a lambda expression:

```
Arrays.sort(rosterAsArray, Person::compareByAge);
```

```
public class Person {

    public enum Sex {
        MALE, FEMALE
    }

    String name;
    Date birthday;
    Sex gender;
    String emailAddress;

    public static int compareByAge
    (Person a, Person b) {
        return
         a.birthday
          .compareTo(b.birthday);
    }
}
```

Method references

```
Arrays.sort(rosterAsArray, Person::compareByAge);
```

The method reference **`Person::compareByAge`** is semantically the same as the lambda expression
`(a, b) -> Person.compareByAge(a, b)`.

Each has the following characteristics:

- Its formal parameter list is copied from **`Comparator<Person>.compare`**, which is **`(Person, Person)`**.
- Its body calls the method **`Person.compareByAge`**.

There are four kinds of method references:

Kind	Example
Reference to a static method	`ContainingClass::staticMethodName`
Reference to an instance method of a particular object	`ContainingObject::instanceMethodName`
Reference to an instance method of an arbitrary object of a particular type	`ContainingType::methodName`
Reference to a constructor	`ClassName::new`

Reference to an Instance Method of an Arbitrary Object of a Particular Type

```
String[] stringArray = { "Barbara", "James", "Mary", "John",
                "Patricia", "Robert", "Michael", "Linda" };
Arrays.sort(stringArray, String::compareToIgnoreCase);
```

The equivalent lambda expression for the method reference **String::compareToIgnoreCase** would have the formal parameter list **(String a, String b)**, where **a** and **b** are arbitrary names used to better describe this example.

The method reference would invoke the method **a.compareToIgnoreCase(b)**.

Reference to a constructor

```
public static <T, SOURCE extends Collection<T>, DEST extends Collection<T>>
    DEST transferElements(SOURCE sourceCollection,
                          Supplier<DEST> collectionFactory) {
        DEST result = collectionFactory.get();
        for (T t : sourceCollection) result.add(t);
        return result;
}
```

The functional interface **`Supplier`** contains one method **`get`** that takes no arguments and returns an object. Consequently, you can invoke the method **`transferElements`** with a lambda expression as follows:

```
  Set<Person> rosterSetLambda =
      transferElements(roster, () -> { return new HashSet<>(); });
```

You can use a constructor reference in place of the lambda expression as follows:

```
Set<Person> rosterSet = transferElements(roster, HashSet::new);
```

The Java compiler infers that you want to create a HashSet collection that contains elements of type Person. Alternatively, you can specify this as follows:

```
Set<Person> rosterSet = transferElements(roster, HashSet<Person>::new);
```

Method references examples

Examples:

FuncInterfaceTutor

FuncInterfaceTask

Data streams

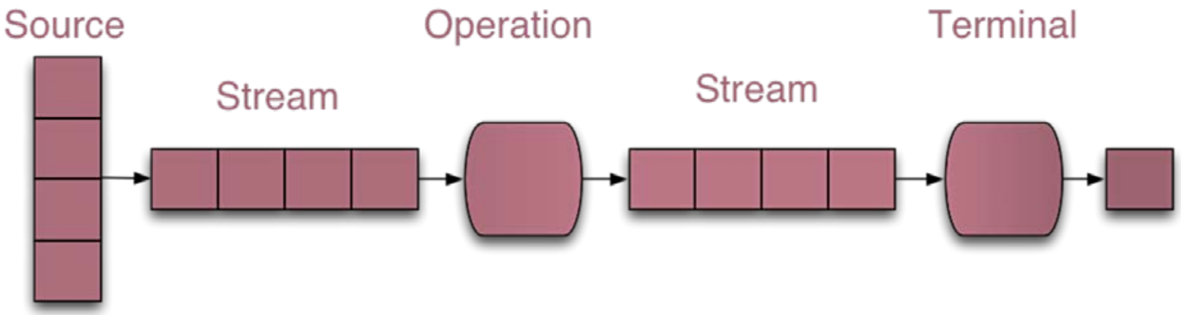
What is a Stream?

Technical answer: a typed interface

```java
public interface Stream<T> extends BaseStream<T, Stream<T>> {
    // ...
}
```

Why streams are *efficient*?

- They may work in parallel, to leverage the computing power of multicore CPUs
- They can be pipelined, to avoid unnecessary intermediary computations



Why a Collection cannot be a Stream?

- The key is the difference between eager and lazy operations. Most operations in the Stream are lazy.

What is a Stream?

- An object on which one can define *operations*

- An object that does not hold any data

- An object that should not change the data it processes

- An object able to process data in « one pass »

- An object optimized from the algorithm point of view, and able to process data in parallel

How to create a stream?

- Using static method Stream.of():

```
Stream.of(1,2,3);
```

- From array

```
String[] arr = {"one", "two", ”three" };
stream = Stream.of(arr);
```

- From collection

```
List<Person> persons;
Stream<Person> stream = persons.stream();
```

- Using generate()

```
Stream<String> stream =
  Stream.generate(() -> "test").limit(10);
```

MAP/FILTER/REDUCE

Map / Filter / Reduce

- Let's take a list of Person :

```
List<Person> list = new ArrayList<>() ;
```

Suppose we want to compute the « average of the age of the people older than 20 »

Lets convert it into stream first…

list.stream()

Map/Filter/Reduce

stream of persons

Person

map((Person p) -> p.getAge())

24 18 34 44 52

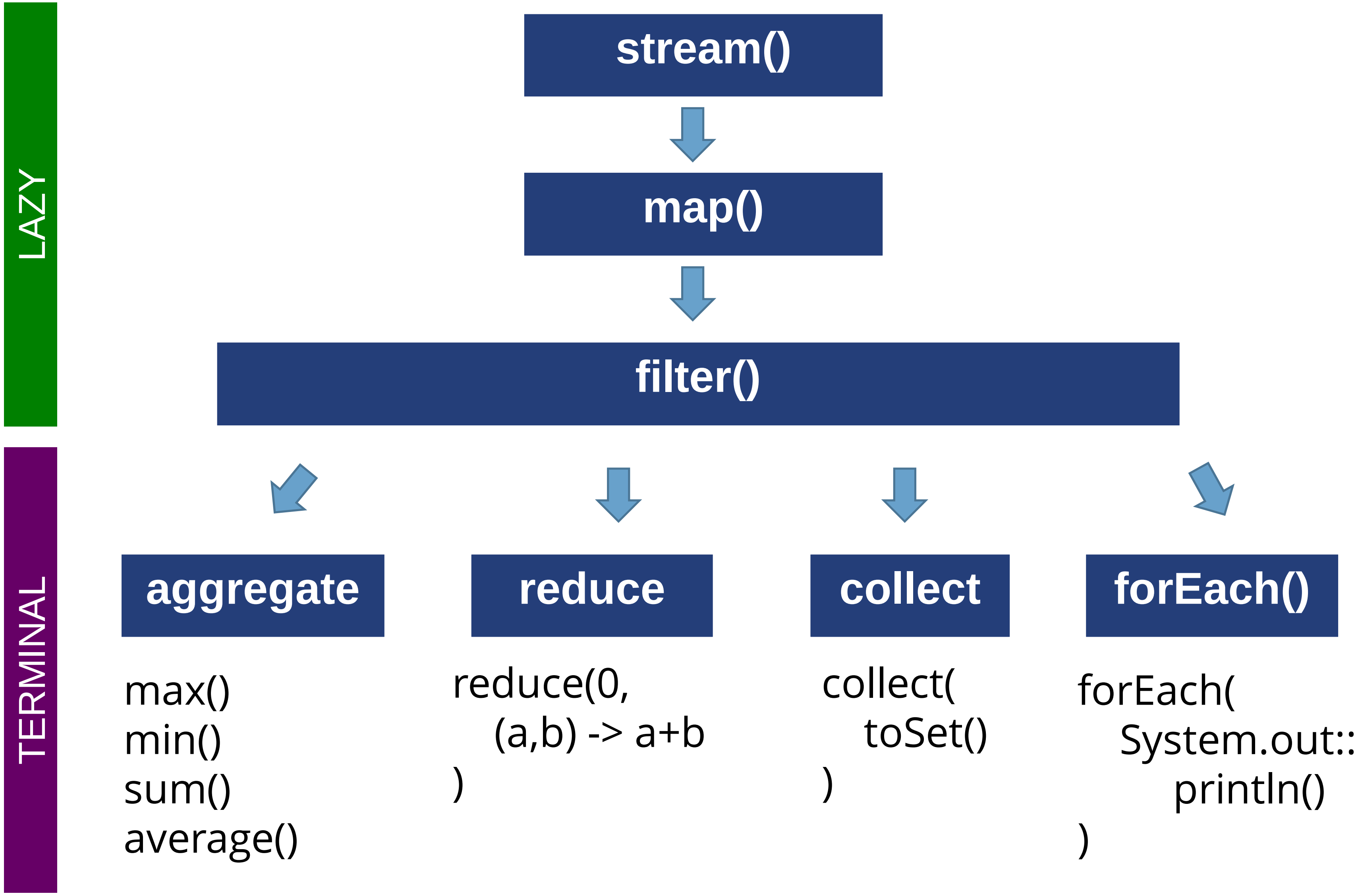
filter(age -> age>20)

24 34 44 52

average()

38.5

Lazy and terminal operations



Aggregation functions

Available aggregations:

- max(), min(), count()

Boolean reductions

- allMatch(), noneMatch(), anyMatch()

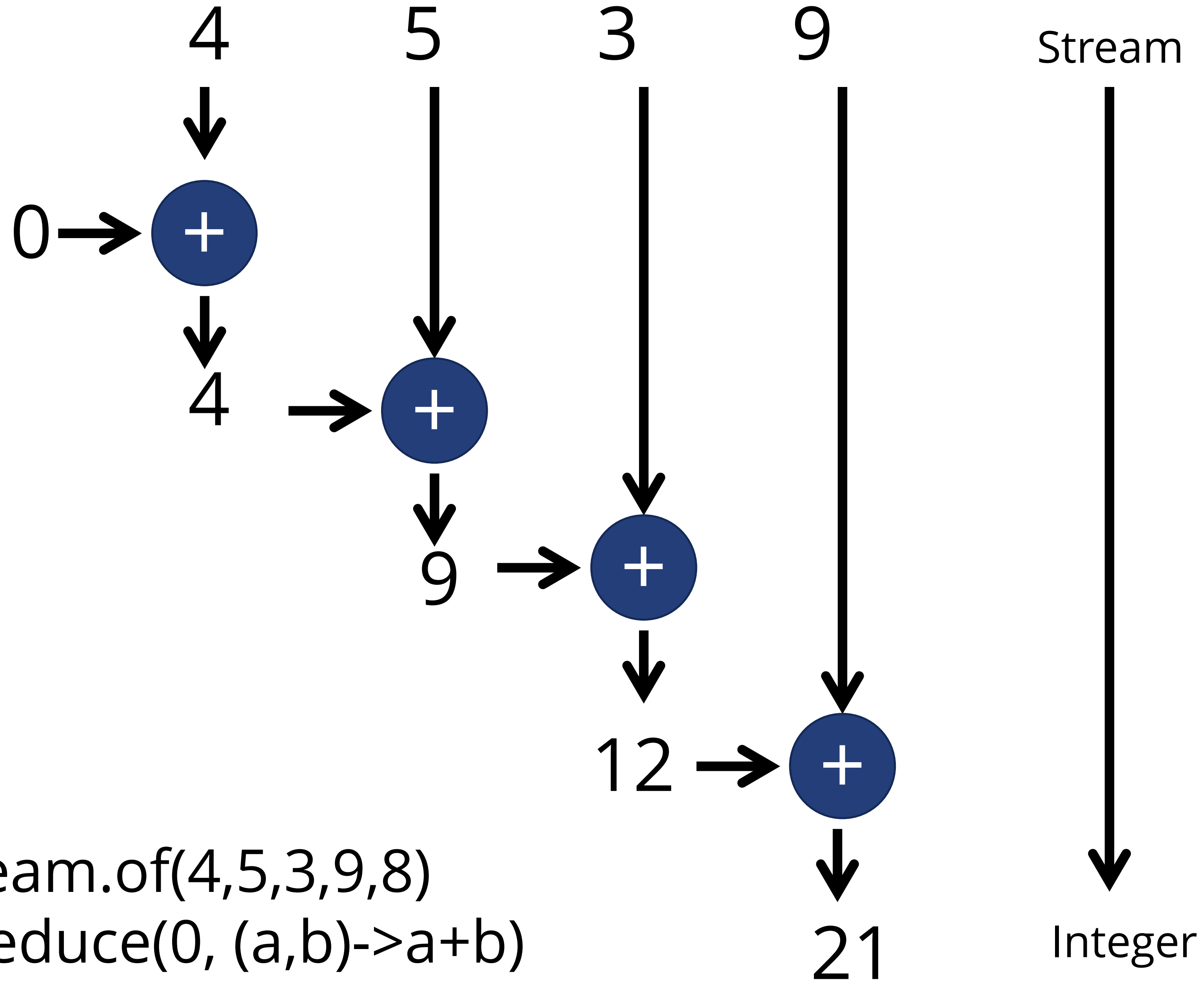
Reductions that return an optional

- findFirst(), findAny()

Reductions are *terminal operations*
They trigger the processing of the data

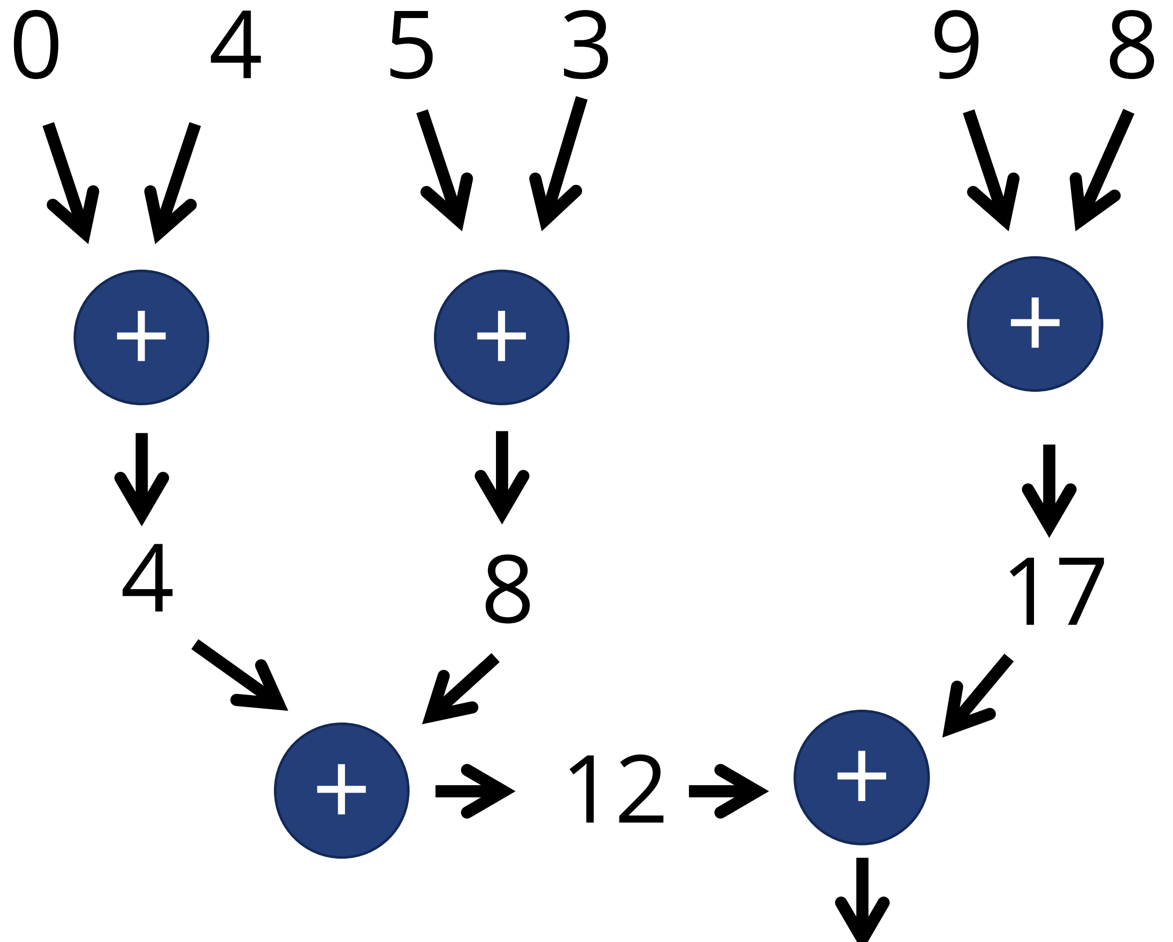
```
persons.map(person -> person.getAge())
    .allMatch(age -> age > 20); // terminal operation
```

Reduce




```
Stream.of(4,5,3,9,8)
  .reduce(0, (a,b)->a+b)
```

Reduce in parallel processing




```
Stream.of(4,5,3,9,8).parallel()
  .reduce(0, (a,b)->a+b)
```

Example:

ReduceTutor

forEach(Consumer)

forEach(Consumer consumer) iterates over all stream elements and apply consumer

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

Consumer<T> is a *functional interface*

It can be implemented by a lambda expression

```java
Consumer<T> c = p -> System.out.println(p);
Consumer<T> c = System.out::println;
Stream.of(1,2,3).forEach(c);
```

Operations on streams

In fact Consumer<T> is a bit more complex

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

    default Consumer<T> andThen(Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> { accept(t); after.accept(t); };
    }
}
```

Consumers may be chained!

Consumers chaining

Example:
ChainConsumers

```
List<String> list = new ArrayList<>();
Consumer<String> c1 = s -> list.add(s);
Consumer<String> c2 = s -> System.out.println(s);
```

```
List<String> list = new ArrayList<>();
Consumer<String> c1 = list::add;
Consumer<String> c2 = System.out::println;
Consumer<String> c3 = c1.andThen(c2);
Stream.of(1,2,3).forEach(c3);
```

Chaining consumers is the only way to have several consumers on a single stream - forEach() does not return anything

Predicates combination

Predicates

```
Predicate<String> p1 = s -> s.length() < 20;
Predicate<String> p2 = s -> s.length() > 10;

Predicate<String> p3 = p1.and(p2);

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

    default Predicate<T> and(Predicate<? super T> other) {
           Objects.requireNonNull(other);
       return (t) -> test(t) && other.test(t);
    }

}
```

Predicate.isEqual()

Predicates

```
Predicate<String> id = Predicate.isEqual(target);
```

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

    static <T> Predicate<T> isEqual(Object targetRef) {
       return (null == targetRef)
               ? Objects::isNull
               : object -> targetRef.equals(object);
    }
}
```

Predicates

```java
List<String> list = new ArrayList<>();
Stream<Person> stream = list.stream();
Stream<Person> filtered =
     stream.filter(person -> person.getAge() > 20);
```

A predicate is taken as a parameter:

```java
Predicate<Person> p = person -> person.getAge() > 20;
```

Predicates

The Predicate interface:

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    default Predicate<T> and(Predicate<? super T> other) { ... }
    default Predicate<T> or(Predicate<? super T> other) { ... }
    default Predicate<T> negate() { ... }
}
Predicate<Integer> p1 = i -> i > 20;
Predicate<Integer> p2 = i -> i < 30;
Predicate<Integer> p3 = i -> i == 0;

Predicate<Integer> p = p1.and(p2).or(p3); // (p1 AND p2) OR p3
Predicate<Integer> p = p3.or(p1).and(p2); // (p3 OR p1) AND p2
```

Warning: method calls do not handle priorities

Predicates

Predicate interface, with static method:

Example:
PredicateTutor

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    // default methods
    static <T> Predicate<T> isEqual(Object o) { ... }
}

Predicate<String> p = Predicate.isEqual("two") ;
Stream<String> stream1 = Stream.of("one", "two", "three");
Stream<String> stream2 = stream1.filter(p.negate()) ;
```

The filter method returns a Stream. This Stream is a new instance.

peek(Consumer)

`Stream peek(Consumer)`

returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.

```
// What does this code do?
List<String> result = new ArrayList<>();
List<Person> persons = ... ;
persons.stream()
       .peek(System.out::println)
       .filter(person -> person.getAge() > 20)
       .peek(result::add);
```

Hint: the peek() method returns a Stream

Summary

- The Stream API defines *intermediary operations*

We saw 3 operations:

- forEach(Consumer) (not lazy)
- peek(Consumer) (lazy)
- filter(Predicate) (lazy)

STREAM MAPPING

Mapping operation

A mapper is modeled by the Function interface with default methods to chain and compose mappings

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);
    default <V> Function<V, R> compose(Function<V, T> before);
    default <V> Function<T, V> andThen(Function<R, V> after);

    static <T> Function<T, T> identity() {
        return t -> t;
    }

}
```

Example

```
UnaryOperator<Integer> f1 = a->a+1;
UnaryOperator<Integer> f2 = a->a*2;
int x = 1;
System.out.println(f1.andThen(f2).apply(x));  // f1(f2(x)) =4
System.out.println(f1.compose(f2).apply(x));  // f2(f1(x)) =3
UnaryOperator<Integer> f3 = UnaryOperator.identity();
print(10, f1);
print(10, UnaryOperator.identity());
print(10, z->z);
```

Flat mapping operation

Method flatMap()

```
<R> Stream<R> flatMap(Function<T, Stream<R>> flatMapper);
<R> Stream<R> map(Function<T, R> mapper);
```

- The flatMapper takes an element of type T, and returns an element of type Stream<R>
- If the flatMap was a regular map, it would return a Stream<Stream<R>>
- Thus a « stream of streams »

- But it is a flatMap!
- Thus the « stream of streams » is flattened, and becomes a stream

Flat mapping operation

Lets calculate the average age of parents

Person Person Person Person

map((Person p) ->
    Stream.of(p.getFather().getAge(),
              p.getMother().getAge())

{48,52} {70,64} {44,44} {52,54}

flatMap(l->l.stream())

48 52 70 64 44 44 52 54

average()

53.5

FlatMapExample

AGGREGATION AND REDUCTION

Reduction step

How does it work?

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream();
Integer sum = stream.reduce(0,
        (age1, age2) -> age1 + age2);
```

- 1st argument: identity element of the reduction operation
- 2nd argument: reduction operation, of type BinaryOperator<T>

BinaryOperator

A BinaryOperator is a special case of BiFunction

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
    //plus default methods
}
```

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, T, T> {
    // T apply(T t1, T t2);
    // plus static methods
}
```

Identity element

The bi-function takes two arguments, so...

- What happens if the Stream is empty?
- What happens if the Stream has only one element?

- The reduction of an empty Stream is the identity element
- If the Stream has only one element, then the reduction is that element

Aggregation

```java
Stream<Integer> stream = ...;
BinaryOperation<Integer> sum = (i1, i2) -> i1 + i2;
Integer id = 0; // identity element for the sum

Stream<Integer> stream = Stream.empty();
int red = stream.reduce(id, sum);

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

Will print:

> 0

Aggregation

```
Stream<Integer> stream = ...;
BinaryOperation<Integer> sum = (i1, i2) -> i1 + i2;
Integer id = 0; // identity element for the sum

Stream<Integer> stream = Stream.of(1);
int red = stream.reduce(id, sum);

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

Will print:

> 1

Aggregation

```
Stream<Integer> stream = ...;
BinaryOperation<Integer> sum = (i1, i2) -> i1 + i2;
Integer id = 0; // identity element for the sum

Stream<Integer> stream = Stream.of(1, 2, 3, 4);
int red = stream.reduce(id, sum);

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

Will print:

> 10

Aggregation: Corner Case

Suppose the reduction is the max

```
BinaryOperation<Integer> max =
   (i1, i2) -> i1 > i2 ? i1 : i2;
```

- The problem is, there is no identity element for the max reduction
- So the max of an empty Stream is undefined…

Aggregation: Corner Case

Then what is the return type of this call?

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream();
...
max = stream.max();
```

- If it is an int, then the default value is 0…
- If it is an Integer, then the default value is null…

Optionals

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream();
...
Optional<Integer> max = stream.max();
```

Optional means « there might be no result »

Optionals

How to use an Optional?

```
Optional<String> opt = ... ;
if (opt.isPresent()) {
   String s = opt.get() ;
} else {

}
```

- The method isPresent() returns true if there is something in the optional
- The method get() returns the value held by this optional

- The method orElse() encapsulates both calls

```
String s = opt.orElse("") ; // defines a default value
```

- The method orElseThrow() defines a thrown exception

```
String s = opt.orElseThrow(()->new MyException("nothing inside"));
```

Example: OptionalTutor

COLLECTORS

Collectors

- There is another type of reduction
- It is called « mutable » reduction
- Instead of aggregating elements, this reduction puts them in a « container »

Collecting in a String

```
List<Person> persons = ... ;

String result = persons.stream()
        .filter(person -> person.getAge() > 20)
        .map(Person::getLastName)
        .collect(Collectors.joining(", "));
```

Result is a String with all the names of the people in persons, older than 20, separated by a comma.

Collecting in a List

```
List<Person> persons = ... ;

List<String> result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(Collectors.toList());
```

Result is a List of String with all the names of the people in persons, older than 20.

Collecting in a Map

```
List<Person> persons = ... ;

Map<Integer, List<Person>> result =
persons.stream()
    .filter(person -> person.getAge() > 20)
    .collect(Collectors.groupingBy(Person::getAge));
```

Result is a Map containing the people of persons, older than 20

- The keys are the ages of the people
- The values are the lists of the people of that age

It is possible to « post-process » the values, with a *downstream collector*

Collecting in a Map

```
List<Person> persons = ... ;

Map<Integer, Long> result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .collect(Collectors.groupingBy(Person::getAge,
                                      Collectors.counting()));
```

Collectors.counting() just counts the number of people of each age.

Examples:

GroupByTutor

CollectorsTutor

SPECIAL STREAM TYPES

IntStream

IntStream - a sequence of primitive int-valued elements supporting sequential and parallel aggregate operations. This is the int primitive specialization of Stream.

```
IntStream.of(2,3,3,4).max();
```

```
List<Integer> numbers = IntStream.range(1, 3)
        .boxed()
        .collect(Collectors.toList());
```

Other primitive values streams are:

- DoubleStream
- LongStream

IntStream usage examples

```
List<String> ls = Arrays.asList(new String[] {"1","2","3"});
OptionalInt ints = ls.stream().mapToInt(Integer::parseInt).max();
int optInt = ls.stream().mapToInt(Integer::parseInt).max().orElse(5);
```

```
// get list of 1 and 2 Integers (to get 1,2,3 use rangeClosed())
List<Integer> numbers = IntStream.range(1, 3).boxed()
        .collect(Collectors.toList());
```

```
OptionalInt max = IntStream.of(5, 10).max(); // 10
```

```
OptionalInt one = IntStream.generate(() -> 1)
        .limit(10).distinct().findFirst(); // 1
```

```
// same as generate, but with a seed -
// will iterate from 0 and for every element will add 3,
// so 0 + 3, 3 + 3 and so on
List<Integer> numbers = IntStream.iterate(0, n -> n + 3).limit(3)
        .boxed().collect(Collectors.toList());
```

```
IntStream first = IntStream.builder().add(10).add(20).build();
IntStream second = IntStream.builder().add(10).build();
IntStream third = IntStream.concat(first, second); // 10,20,10
```

IntStreamTutor

Random

Random is used to generate a stream of pseudorandom numbers.

- If two instances of Random are created with the same seed, and the same sequence of method calls is made for each, they will generate and return identical sequences of numbers.
- Instances of Random are **threadsafe**. However, the concurrent use of the same Random instance across threads may encounter **poor performance**. Consider instead using **ThreadLocalRandom** in multithreaded designs.
- Instances of Random are **not cryptographically secure**. Consider instead using **SecureRandom** to get a cryptographically secure pseudo-random number generator

Example:

RandomTutor

Parallel streams

`Stream parallel()`
returns an equivalent stream that is parallel

`Stream unordered()`
returns an equivalent stream that is unordered. May return itself, either because the stream was already unordered, or because the underlying stream state was modified to be unordered.

Examples:
- ParallelTest
- ForkJoinSum
- ForkJoinFreq

Thank You!

think.
create.
accelerate.