Udemy Course

Understanding Functional Interfaces
A functional interface is an interface with only a single abstract method, such that when we instantiate it we can give a single lambda, anonymous definition or method reference to “implement” the abstract method.

Type Inference for Lambda
How is type inference used for lambda expressions?

- Inferring the functional interface
  - E.g. new Thread(lambda) infers that lambda is a Runnable as the constructor for Thread takes in a Runnable
- Inferring the argument type
  - E.g. Function<Integer, Integer> square = x -> x*x;
  - Infers that x is an integer based on LHS
  - Infers that the return type is also an integer based on LHS

When does type inference work for lambda expressions?

- Type inference occurring on the RHS of the expression
  - Consumer<String> = lambda
- When using a lambda expression as a parameter, and the inference is from the parameter type
  - new Thread(lambda)
- Returning a lambda, where the type is inferred from the return type
  - return lambda
- From type casting
  - (Consumer<String>) lambda

However, when using wildcard operators, the same issues will pop up:

```
Consumer<?> c = msg -> System.out.println(msg.length());
// does not compile as <?> is inferred to be of type Object (due to lack of
information), and an Object does not have the method length. It is supposed
print a String.

Consumer<?> c = (String msg) -> System.out.println(msg);
// this will work
```

Type inference also depends on the amount of information available:

```
// Standard syntax
Consumer<String> c1 = msg -> System.out.println(msg.length());

// Compile-time error: not enough info - doesn't know x1 is a consumer and
msg is a string
Object x1 = msg -> System.out.println(msg.length());
```

```
// Compile-time error: not enough info - doesn't know x2 is a consumer
Object x2 = (String msg) -> System.out.println(msg.length());

// OK: cast added
Object x3 = (Consumer<String>) ((String msg) ->
System.out.println(msg.length()));
```

Variable Capture
Variable capture will only fail for local variables that are not effectively final. It does not matter for instance and static variables.

Method References
Method references have no intrinsic type, so they need to be called in a context with enough information. This is similar to lambda expression. The contexts where this works is the same as for lambda expressions.

```
Supplier<Thread> s1 = Thread::currentThread;
System.out.println(s1.get());
// this will print the reference to the current thread
// notice that the currentThread method takes in no arguments (same as
supplier) and returns a Thread. This is however unknown as method references
have no intrinsic type, and the compiler only knows of it because of the
Suppler<Thread>.
```

However, we can also do it in terms of return type. Let us create our own functional interface. Remember that the functional interface only has one abstract method.

```
interface ThreadSupplier {
    Thread giveMeAThread();
}
```

We can also instantiate a function like so:

```
ThreadSupplier ts = Thread::currentThread;
System.out.println(ts.giveMeAThread());
```

Once again, the compiler only knows that the Thread::currentThread is returning a Thread object thanks to the return type declaration in the interface.

Method references need not refer to static methods. We can also refer to instance methods with instance unspecified, instance methods with instance specified, or even a constructor! Less common are super class methods and array constructors.

For example:

```
Employee frank = new Employee("Frank", 3000);

// instance method with instance specified
Supplier<Integer> s2 = frank::getSalary;
System.out.println(s2.get()); // 3000
```

```
// instance method with instance not specified
Function<Employee,Integer> f1 = Employee::getSalary;
System.out.println(f1.apply(frank)); // 3000

// Even this is an instance method with instance specified, where the
instance is System.out
Consumer<String> c1 = System.out::println;
```

There is also a shortcut to creating comparators:

```
Comparator<Employee> byName = Comparator.comparing(Employee::getName);
```

The default method comparing of the comparator interfaces actually takes in a method that can extract a key from items to be sorted, and returns a comparator object. The type inference occurs from the LHS.

Below is a way to print objects in an array. Read and understand:

```
public static <T> void printAll(T[] array,Function<T,String> toStringFun) {
    int i = 0;
    for (T t: array) System.out.println(i++ + ":\t" + toStringFun.apply(t));
}
public static void main2() {
    Employee dept[] = new Employee[5];
    dept[0] = new Employee("Alec", 1500);
    dept[1] = new Employee("Bob", 1600);
    dept[2] = new Employee("Claire", 1700);
    dept[3] = new Employee("Danielle", 1800);
    dept[4] = new Employee("Ethan", 1900);
    printAll(dept, Employee::getName);
    System.out.println("");
    // Compile-time error: type inference failure
    // printAll(dept, Employee::getSalary);
    printAll(dept, emp -> "" + emp.getSalary());
}
```

Functional Interfaces before Java 8

- Runnable
- Callable
- ActionListener
- Comparable (Opposite of pure functional interface, as comparison can only take place when the objects have something to compare i.e. state)
- Comparator (The only pure functional interface!! As it is stateless)

Runnable

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

Usually used as the body of a thread:

```
public Thread(Runnable r) // constructor
```

Passing this into the thread constructor, a new thread will start and it will start from the run method of the Runnable object. Its run method is the “entry point” of a new thread.

Callable

```
interface Callable<V>{
    V call();
}
```

Usually used to get a value from the body of a thread. However, it cannot be used directly with the Thread class, and will need to be used together with the following method of the ExecutorService interface:

```
<V> Future<V> submit(Callable<V> task)
```

It returns a Future object which represents the eventual value returned by the Thread when it is completed.

ActionListener

```
interface ActionListener{
    void actionPerformed(ActionEvent event);
}
```

This is usually used to provide callback functions for GUI events. This is used, for example, with the class Jbutton:

```
public void addActionListener(ActionListener l)
```

Comparable

```
interface Comparable<T> {
    int compareTo(T other);
}
```

This is used together with Collections.sort, where if the elements in the list implement Comparable, then they can be compared and sorted.

```
public static <T extends Comparable<? super T>> void sort(List<T> List)
```

Comparator

```
interface Comparator<T>{
    int compare(T first, T second);
}
```

This comparator represents an external ordering criterion for a class, and is used with the following method from the Collections class:

```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

New Functional Interfaces
java.util.function
Over forty new functional interfaces categorised into six families

These are the six main types and the most important functional interface/representative functional interface of each type:

Function Type	Functional Interface Name
nothing -> T	Supplier
T -> nothing	Consumer
T -> T	UnaryOperator
T, T -> T	BinaryOperator
S -> T	Function
T -> boolean	Predicate

Consumer

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

Accepts an object

Supplier

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

Supplies an object

Example of usage:
Let's say we are using Object.requireNonNull(T t, String message), which checks if the object t is a null input or not. However, we actually run a very expensive method to return the correct message.

This results in the expensive method being called or evaluated eagerly for every parameter needed to be check, regardless of whether it is a null value or not. We can reduce the overhead by using the overloaded version of this function, Object.requireNonNull(T t, Supplier<String> msgSupplier). This method will only call for the msgSupplier.get() when the value t being passed is actually null, hence saving on the computation costs.

UnaryOperator

```
interface UnaryOperator<T>{
    T apply(T t);
}
```

An object updater/modifier which preserves the type of the object

Usually it is used with the following method from the List class:

```
void replaceAll(UnaryOperator<T> op)
```

BinaryOperator

```
interface BinaryOperator<T>{
    T apply(T a, T b);
}
```

An operation between two objects, preserving the type

It used usually used in the reduce operation for streams:

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

Function

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

An object transformer, from one type to another

A function can be used as a key extractor for comparators, as seen before in:

```
Comparator<Employee> byName = Comparator.comparing(Employee::getName);
// OR
Comparator<Employee> byName = Comparator.comparing(e -> e.getName());
```

It can also be used for the stream mapping.

Predicate

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

It tests whether an object has a certain desired property.

One way to use it is with the Collections interface:

```
boolean removeIf(Predicate<T> p)
```

It is also used in the filter operation in streams

Primitive Functional Interfaces
There are functional interfaces for primitive types as well! The purpose is to reduce the overhead that comes with wrapping and unwrapping.

Let us look at some examples:

```
interface IntBinaryOperator{
    int applyAsInt(int a, int b);
}
interface LongBinaryOperator{
    long applyAsLong(long a, long b);
}
interface DoubleBinaryOperator{
    double applyAsDouble(double a, double b);
}
IntBinaryOperator sum = (a, b) -> a+b;
```

There are specialised types of functional interfaces for all of these 3 types:

FI Name	X = Int, Long or Double	Function Type
Supplier	XSupplier	nothing -> X
Consumer	XConsumer	X -> nothing
UnaryOperator	XUnaryOperator	X -> X
BinaryOperator	XBinaryOperator	X, X -> X
Predicate	XPredicate	X -> boolean

However, for Function it can be more complicated:

FI Name	Function Type	Examples
ToXFunction<T>	T -> X	ToIntFunction<T>
XToYFunction	X -> Y	IntToLongFunction IntToDoubleFunction

There is no IntToIntFunction because that would just be the IntUnaryOperator!

Lastly, we have one functional interface for booleans:

```
interface BooleanSupplier{
    boolean getAsBoolean();
}

final Random rand = new Random();
BooleanSupplier randomBS = rand::nextBoolean;
```

We actually also have things such as Comparator.comparingX

Combining Functions
Through composing functions, we can combine simpler functions to create complex ones. There are 4 functional interfaces that allow this:

- Predicate
- Consumer
- Function
- Comparator

Predicate
Let's say we have two predicates a and b.

```
Predicate<T> a = //define here;
Predicate<T> b = //define here;
Predicate<T> c = a.and(b); // &&
Predicate<T> d = a.or(b); // ||, short circuits
Predicate<T> e = a.negate(); // !
```

Consumer
Apply two consumers to the same argument sequential composition

```
Consumer<T> a = //define here;
Consumer<T> b = //define here;
```

```
Consumer<T> c = a.andThen(b);
```

An example would be:

```
PrintWriter writer = new PrintWriter("filename.txt");
Consumer<String> logger = writer::println;
Consumer<String> screener = System.out::println;
Consumer<String> both = screener.andThen(logger);
both.accept("Program started");
// prints to screen first then to the file
```

Functions
Can combine two functions sequentially. Given f: A->B and g: B->C
f.andThen(g) -> f(g(x)) type A->C
g.compose(f) -> g(f(x)) type A->C

Example:

```
Function<Employee, String> getName = Employee:getName;
Function<String, Character> getFirstLetter = name -> name.charAt(0);
Function<Employee, Character> initial = getName.andThen(getFirstLetter);
// returns first character of an employee's name
```

Comparators
This allows us to apply two comparators lexicographically

- Apply first comparator
- If first comparison is inconclusive, apply second comparator

Example below:

```
Comparator<Employee> byName = Comparator.comparing(Employee::getName);
Comparator<Employee> bySalary =
Comparator.comparingInt(Employee::getSalary);
Comparator<Employee> byNameAndSalary = byName.thenComparing(bySalary);
```

Notice that we are using Comparator.comparingInt!!

Streams
A powerful framework for sequential data processing. It is a sequence of objects that support a special type of operation called **internal iteration.**

Streams
What is the difference between streams, lists and iterators?
Lists

- add
- remove

- search
- scan/iterate

Iterator

- scan/iterate

Stream

- internal iteration

Let's see the differences when printing a sequence of objects:

```
List<Integer> l = // something;
for (Integer n:l)
    System.out.println(n);

Iterator<Integer> i = // something;
while(i.hasNext())
    System.out.println(i.next());

Stream<Integer> s = // something;
s.forEach(System.out::println);
```

Notice that the stream has no reference to any elements! There is no explicit looping either. This is internal iteration. In other words, we just need to pass a certain function to the stream, and it will take care of it for us internally.

java.util.Stream<T>

- 30+ instance methods
- 7+ static methods

Its core functionalities lie in repeatedly applying an operation to all elements, and shifting the responsibility of doing so from the client to the stream library.

Types of Stream Operations

- Build operations - create a stream from a data source
- Intermediate operations - convert one stream into another
- Terminal operations - create a stream into something else or nothing (void)

Build -> Intermediate -> Intermediate -> … -> Intermediate -> Terminal

Let's say we want to print the names of employees with salary above $2500 in alphabetical order.

Using a list, we will need to filter, sort, then iterate and print. However, with streams, we

just need to:

```
Employee[] emps = {/*something here*/};
Arrays.stream(emps).filter(e->e.getSalary()>=2500)
                   .map(Employee::getName)
                   .sorted()
                   .forEach(System.out::println);
```

- Arrays.stream is a build operation
- Filter, map and sorted are intermediate operations
- forEach is a terminal operation (side effect of printing)

Ordered vs Unordered Streams
Objects in a stream may have a fixed order or not

- This is also known as the encounter order

Ordered stream has the operations applied on it performed in the encounter order. Unordered stream may have its operations performed in any order, with no guarantee on the order.

Sequential vs Parallel Streams
A sequential stream has its operations performed on one object at a time. A parallel stream may have its operations performed on several objects in parallel.

Stream Building
There are 3 ways to build a stream:

- Static sequence of objects
- Collection of objects
- Computation

Static Sequence
This uses the static method Stream.of

```
public static <T> Stream<T> of(T… values)
Stream<Integer> fib = Stream.of(1,1,2,3);
Stream<String> italianNumbers = Stream.of(“uno”, “due”, “tres”);
```

This can also be used on an array:

```
Employee[] emps = // something
Stream<Employee> empStream = Stream.of(emps);
```

Collections
The above is one way to do it, but there are more specific methods on the Arrays class the help with the conversion.

```
public static <T> Stream<T> stream(T[] array)
```

```
public static <T> Stream<T> stream(T[] array, int startInclusive, int
endExclusive)

Stream<Employee> empStream = Arrays.stream(emp);
```

Streams from arrays are ordered and sequential!! But there are ways to convert them to unordered and parallel.

There are also ways to stream from a Collections object! In the Collection<T> interface:

```
Stream<T> stream()
Stream<T> parallelStream()

Collection<Employee> emps = // something
Stream<Employee> empStream = emps.stream();
```

For collections, if you start from an ordered collection such as lists, you get ordered results. However, if you start from unordered collections such as sets (unless the set is sorted like TreeSets), it is unordered.

Computation
This is done through lazy evaluation, where elements are computed on demand:

- Each element is calculated separately (via Supplier), or
- Each element is calculated from the previous one (via UnaryOperator)

This is also what is known as Infinite Streams, as computed streams are potentially infinite, although a finite number of them will ever be created.

Using Supplier
In the interface Stream, there is a static method:

```
static <T> Stream<T> generate(Supplier<T> s)
```

This generates an unordered infinite stream. Recall that a supplier is an object with a single function that takes no argument and returns an object. This needs not to be a Supplier object. For example:

```
Random random = new Random();
Stream<Integer> randoms = Stream.generate(random::nextInt);
```

and we get an unordered infinite stream.

Using UnaryOperator
Let us try to generate a stream using a UnaryOperator

```
static <T> Stream<T> iterate(T first, UnaryOperator<T> next)
```

This generates an ordered infinite stream, with the next element building onto the previous

element. For example:

```
Stream<String> as = Stream.iterate(“a”, s -> s + “a”);
```

There are many other ways to create a stream. Can check out the static methods of stream.

Lazy Evaluation
Lazy evaluation is also known as on-demand evaluation. The evaluation happens as late as possible. This is in comparison with eager evaluation, which happens as early as possible.

Example:

```
Random random = new Random();
Stream<Integer> randoms = Stream.generate(random::nextInt);
```

The integers will only be generated when required. And when is that? It will only be generated during the terminal operation.

Elements are created and processed when required by the terminal operation, that is, elements are pulled from the end and not pushed from the start.

A full example:

```
final Random random = new Random();
Supplier<Integer> supp = () -> {
    Integer result = random.nextInt();
    System.out.println(“(supplying “ + result + “)”);
    return result;
};
Stream<Integer> randoms = Stream.generate(supp);
randoms.filter(n-> n>=0).limit(3).forEach(System.out::println);
```

The limit operator helps to stop the generation process. This goes on until 3 non-negative integers have been generated, then the terminal operation is applied.

On the other hand, we can have this:

```
Stream<Integer> randoms2 = Stream.generate(supp);
randoms2.limit(3).filter(n->n>=0).forEach(System.out::println);
```

This one will only generate 3 integers due to the limit, before filtering out the non-negative. Hence the number of integers that may possibly be printed is 0<=n<=3.

Traverse Once
A stream can only be traversed once. If we try to traverse it again, we get a java.lang.IllegalStateException. This is because the stream has already been operated upon or closed, which happens when final operation is executed (aka the final stream that did not get its return value used!)

This emphasises the importance of a proper stream pipeline. This is because any intermediate operation actually returns a new stream, which is then immediately operated upon by the following operation. If we split up the two operations e.g.

```
randoms.limit(2);
randoms.forEach(System.out::println);
// ERROR
```

We get the same error, that the stream has been closed.

This is because the limit method, being an intermediate operation, closes the current stream and returns a new one. If we must split the operations, we can use:

```
Stream<Integer> newStream = randoms.limit(2);
newStream.forEach(System.out::println);
```

This will work.

However, some functions such as forEach is a void function, and will not actually return anything. Hence those are really terminal operations.

Streams as Monads
First of all, let us look at the flatMap operation in interface Stream<T>:

```
// simplified
<R> Stream<R> flatMap(Function<T, Stream<R>> f)
```

It basically applies function f on every element in a stream and converts it into a stream of type R. The flatMap function then merges everything into a single stream of type R.

Let's say we need to convert a set of books, each of which is a list of words, into a stream of distinct words. We can use flatMap:

```
Set<Book> lib = // something;
lib.stream()
   .flatMap(book->book.getWords().stream())
   .distinct()
   .forEach(System.out::println);
```

Monad
A monad is a parametric type M<T> that helps to add some context to T. This usually is some kind of extra information.

It also has two main operations:

- Unit: T -> M<T>
- Bind: (m: M<T>, f: T->M<R>) -> M<R> (what?!)

Let us see.

Unit(t: T): M<T> basically wraps a plain value in some default context. With streams, the way to do it is the Stream.of(…) method.

For example, we can wrap a String into a Stream<String>:

```
Stream<String> s = Stream.of(“Hello”);
```

The Bind operation takes in two inputs:

- A value with context m: M<T>
- A function f: T-> M<R> from plain values to another type with its own context

And what it does is to apply f to m. Of course, we cannot simply apply it, as it requires a plain value, but our T is wrapped in context m. Hence, there are 3 steps to the process:

- Unwrap m to get T and a context
- Apply f to T to get M<R>
- Combine the old context and new context to get another M<R>

Therefore, bind **is the recipe for combining contexts.**

Let us see an example.
With stream.flatMap(f), what we have is:

- Unwrap stream
- Apply f
- Concatenate resulting streams

Streams are therefore Monads!!

But why do we care so much about streams being monads? This is because monads support operation chaining.

Given f: T->M<R> and g: R->M<S>

We can get:
T —unit—> M<T> —bind with f—> M<R> —bind with g—> M<S>

This is the same as calling flatMap twice, with each time changing the stream that you’re working with slightly.

Then how about map?

Let’s say we have:

- Value with context m: M<T>
- Function f with f: T->R (knows nothing about contexts)

However, we cannot apply bind as bind is supposed to return a M<R>. But we can teach contexts to f:

- Let f'(t) = unit(f(t)), where f':T->M<R> (f then default context)
- Bind(m, f') -> M<R>

In streams, this is called map.

Stream Operations
There are two types of operations:

- Intermediate: return Stream<T> or Stream<R>
- Terminal: return anything else or void

Filter
This is a category of intermediate operations that select some elements and discard others based on:

- Content: filter, takeWhile, dropWhile
- Amount: limit
- Uniqueness: distinct

All methods here are instance methods.

filter method:

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

Discards all elements which violate a Predicate. Since a wildcard is used, we can also use a predicate for Person to filter out Employee elements.

limit method:

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

Picks the first n elements or less.

distinct method:

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

Discard duplicates according to the equals() method.

Example of the above methods being used together:

```
final Random rand = new Random();
Stream<Integer> randoms = Stream.generate(rand::nextInt);
```

```
randoms.filter(n->n>0)
       .distinct()
       .limit(10)
       .forEach(System.out::println);
```

Swapping distinct and limit: We might end up with less than 10 numbers
Swapping distinct and filter: We spend time removing duplicate negative numbers

The distinct method is one of the few stateful intermediate operations, as it does not operate independently on each element. It is thus harder to parallelise.

takeWhile method:

```
Stream<T> takeWhile(Predicate<? super T> property)
```

Takes elements while Predicate is true, until the first false, then it discards the rest, regardless of whether the Predicate will be true for the remaining elements.

dropWhile method:

```
Stream<T> dropWhile(Predicate<? super T> property)
```

It's the reverse of takeWhile. While the predicate is true, we keep discarding the elements, until the first element which returns false. It will be the first element. The rest will all be taken regardless of whether the Predicate will be true for them.

Suppose we have a sequence of employees sorted by increasing salary, and we want the employees with salary of at most 2000. We can do this the traditional way:

```
Stream<Employee> sortedEmps = // something;
sortedEmps.filter(e -> e.getSalary() <= 2000)
          .forEach(System.out::println);
```

This however scans the whole stream. We can make it more efficient with:

```
sortedEmps.takeWhile(e-> e.getSalary()<=2000)
          .forEach(System.out::println);
```

This will stop as soon as a salary >2000 appears. This is much more efficient.

However, both takeWhile and dropWhile provide no guarantee as to what elements would be taken or dropped when used on an unordered stream. This is almost useless, but it may be useful if we want to stop a stream according to some external condition.

```
static volatile boolean keepGoing = true;
infiniteStream.takeWhile(x->keepGoing)
```

This is however very non-functional, since this is not stateless. SHOULD BE AVOIDED.

Transform and Rearrange

Intermediate operations that:

- Transform the type of elements: flatMap, map
- Sort the elements: sorted
- Drop the order: unordered

flatMap & map:

```
function T->Stream<R>: flatMap
function T->T or T->R: map
```

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

Due to the wildcard operators, the function can take in a superclass of T and return a subclass of R.

Let's say we want to find distinct cities where at least one employee lives:

```
Stream<Employee> emps = // something;
emps.map(Employee::getAddress) //Stream<Address>
    .map(Address::getCity) //Stream<City>
    .map(City::getName) //Stream<String>
    .distinct()
    .forEach(System.out::println);
```

sorted:
Sorting:

- Based on natural order (Comparable)
- OR based on comparator

```
Stream<T> sorted()
Stream<T> sorted(Comparator<? super T> comp)
```

These are stateful operations, means they do not operate independently on each element. This has consequences on parallel stream.

Example of finding names of 10 employees with highest salary:

```
emps.sorted(
    Comparator.comparingInt(Employee::getSalary)
              .reversed()
).limit(10)
 .map(Employee::getName)
 .forEachOrdered(System.out::println);
// forEachOrdered is a forEach that respects the encounter order
```

unordered:
Unordering:

- Convert stream to unordered
- Release ordering guarantees

```
Stream<T> unordered()
```

It does absolutely nothing to the data. It simply releases ordering guarantees, such that any operations on the stream need not to respect the encounter order anymore. This can improve parallel performance.

For example:

```
List<Integer> list = //something;
long n = list.parallelStream() //ordered from list
             .unordered() // unordered
             .distinct() // significant speedup
             .count();
```

Terminal Operations:
Terminal Operations can:

- Extract a single element - findFirst, findAny, max, min
- Put all elements in an array - toArray
- Count the elements - count
- Check a condition - allMatch, anyMatch, noneMatch
- Execute arbitrary code on each element - forEach, forEachOrdered, peek

We can also partition them according to the return type:

- void: forEach, forEachOrdered, peek (actually an intermediate operation!)
- boolean: allMatch, anyMatch, noneMatch
- array: toArray
- long: count
- T: findFirst, findAny, max, min

void operations

```
void forEach(Consumer<? super T> c)
void forEachOrdered(Consumer<? super T> c)
Stream<T> peek(Consumer<? super T> c)
```

forEach - ignores the encounter order
forEachOrdered - respects the encounter order
peek - it applies the consumer on every element, but returns the stream as it is before performing the consumer

peek is usually used for debugging as we peek at the stream using println before we continue the processing with other operations. peek will not actually run until another terminating condition is performed.

boolean operations

```
boolean allMatch(Predicate<? super T> property)
boolean anyMatch(Predicate<? super T> property)
boolean noneMatch(Predicate<? super T> property)
```

allMatch - is the property true for all elements? needs to scan whole stream
anyMatch - is the property true for at least one element? stops after the first true.
noneMatch - is the property false for all elements? needs to scan whole stream

Example to check if all employees have a valid name:

```
boolean allValid = emps.stream()
                       .allMatch(
                        e.getName()!=null&&
                        e.getName().length()>0);
```

array operations

```
Object[] toArray()
<A> A[] toArray(IntFunction<A[]> factory)
```

First toArray returns an array of objects as it is not aware of the runtime type of elements due to erasure
Second toArray provides array of the right type, where A=T. However, a factory of arrays of the right type is needed.

Example for finding employees with low salaries:

```
Object[] lowEmps = emps.filter(e->e.getSalary()<2000)
                       .toArray();
```

However, we end up with an array of objects. If we want to create arrays of employees:

```
Employee[] lowEmps = emps.filter(e->e.getSalary()<2000)
                         .toArray(Employee[]::new);
```

The factory method is the standard allocator of arrays.

long operations
count, which we've seen before.

T operations
Finally, we have terminal operations that can extract elements from the stream.

```
Optional<T> findAny()
Optional<T> findFirst()
Optional<T> min(Comparator<? super T> comp)
```

```
Optional<T> max(Comparator<? super T> comp)
```

findAny returns an arbitrary element, no guarantee.
findFirst returns the first element in the encounter order (for an ordered stream), else no guarantee.
min returns the minimum element according to comp
max returns the maximum element according to comp

All of them return an Optional of type T, which may or may not contain a value. This is because a stream may be empty, but a return type is still required. The proper functional way of returning a potentially null value is using an Optional.

Optionals
It is the functional alternative to returning null, and it can be chained into an intuitive and readable pipeline such as:

```
String result = stream.min(comp).orElse("default");
```

So the optional class is also a Monad!

Methods of Optional:

```
boolean isPresent() // is there an object inside?
T get() // unwraps object or throw NoSuchElementException

flatMap // monad
map
filter
of // static factory
```

Example:

```
Collection<String> strings = //something;
Optional<String> longest = strings.stream()
    .max(Comparator.comparingInt(String::length));
```

Reductions and Collectors
Custom terminal operations which can be classified into two types:

- Reduce operations
    - Functional/Stateless terminal operations
- Collect operations
    - Mutable/Stateful

In both cases, we are talking about summarising a stream into a single object.

Reduction
A common summarisation pattern is to repeatedly apply a binary operation, starting from a seed:

```
T summary = seed;
for (T t: collection){
    summary = operation(summary, t);
}
```

Examples:

- Sum the elements
- Compute min/max

We can get the same reason using:

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

For example, string concatenation:

```
String allWords = words.stream()
                       .reduce(“”,(a,b)->a+””+b);
```

This is inefficient! Quadratic complexity. Better to use a mutable summarisation.

Parallel Reductions?
Reduction can be efficiently paralleled, provided:

- The binary operation is stateless and associative
- The seed is an identity of the binary operation

Both must be true for it to work!

Binary operations must be stateless:

- This means that operations should only depend on its arguments and nothing else.
- E.g. (int a, int b) -> a+b;
- Stateful: if (sum>0) return a; else return b; where sum is an external value.

Binary operations must be associative:

- Order of application is irrelevant
- Addition, multiplication, string concatenation
- Not associative: subtraction

Seed must be identity:

- It does not affect the result of the operation
- Op(seed, a) = a
- Addition: seed = 0
- Multiplication: seed = 1

- String::concat: seed = “"

Mutable Summarisation
Without using streams, we might do this using a summary object:

```
Summary summary = new Summary();
for (T t: collection){
    summary.update(t);
}
```

For example, we can put elements into another collection, or build a (key, value) map.

This correspond to collect for Streams:

```
Summary summary = collection.stream()
    .collect(()-> new Summary(),
        (Summary s, T t) -> s.update(t),
        combiner);
```

They are, respectively:

- factory (Supplier<S>) - provides initial summary object
- accumulator (BiConsumer<S, T> -> nothing) - updates summary with new element
- combiner (BiConsumer<S, S> -> nothing) - incorporates second summary into the first (only used if parallel)

Both the accumulator and combiner need to be stateless and associative, else running it in parallel may result in it not working properly.

Return type of collect is a summary of type S while we are dealing with a stream of type T.

Example of String building:
This is the normal way of doing it:

```
StringBuilder summary = new StringBuilder();
for (String s:collection){
    summary.append(s);
}
```

With streams, we can have:

```
StringBuilder summary = collection.stream()
    .collect(()->StringBuilder(),
             (StringBuilder builder, String s) -> builder.append(s),
             (StringBuilder builder1,
              StringBuilder builder2) -> builder1.append(builder2));
```

SHORTENED:

```
StringBuilder summary = collection.stream().collect(
```

```
    StringBuilder::new, //factory
    StringBuilder::append, //accumulator
    StringBuilder::append); //combiner
```

The two append methods are actually different, as the methods are overloaded.

EVEN SHORTER:

```
String summary = collections.stream().collect(Collectors.joining());
```

Collectors

```
<S,R> R collect (Collector<T,S,R> collector)
```

T: type of elements of current stream
S: type of summary
R: type of result (may be same as summary! May also be different due to the finisher)
An interface that incorporates factory, accumulator and combiner PLUS

- Finisher: optional final step from summary to result
- Characteristics: declaring ordered/unordered, sequential/concurrent

Standard Collectors

- Strings: joining
- Standard Collections: toList, toSet, toCollection
- Map: toMap, groupingBy, partitioningBy

Strings

```
Collectors.joining() // concatenates with no delimiter
Collectors.joining(CharSequence delimiter) // with delimiter
```

Collections

```
<C extends Collection<T>> toCollection(Supplier<C> factory) // summarises
into C
toList() // summarises into List<T>, no guarantee on kind e.g. ArrayList or
LinkedList?
toSet() // summarises into Set<T>, no guarantee on kind
```

Example, sorting employees in a stream by salary:

```
TreeSet<Employee> tree = emps.collect(
    Collectors.toCollection(
        ()-> new TreeSet<Employee>(
            Comparator.comparingInt(Employee::getSalary)))); //Supplier of
TreeSets
```

Maps
toMap -> each element generates a key, value pair: Map<K, V>
groupingBy -> elements are grouped based on a Function: Map<K, List<T>>
partitioningBy -> elements are grouped based on a Predicate: Map<Boolean, List<T>>

To use toMap, we need:

- keyMapper: from element to key K
- valueMapper: from element to value V

```
toMap(Function<T,K> keyMapper, Function<T,V> valueMapper)
toConcurrentMap(Function<T,K> keyMapper, Function<T,V> valueMapper)
```

toMap returns a sequential Collector, while toConcurrentMap returns a concurrent but unordered Collector.

They throw an error if two elements map to the same key. No guarantees on actual type of map (e.g. TreeMap or HashMap etc.) and the argument types above are simplified for clarity.

Example, to build a map from names to salaries:

```
Map<String, Integer> salaries = emps.collect(
    Collectors.toMap(Employee::getName,
                     Employee::getSalary));
```

To use groupingBy, we need:

- keyMapper

```
groupingBy(Function<T,K> keyMapper)
groupingByConcurrent(Function<T,K> keyMapper)
```

Similarly, groupingBy returns a sequential Collector while groupingByConcurrent returns a concurrent but unordered Collector.

No guarantees on actual type of map nor list. Argument types are also simplified for clarity.

Example, to group employees by salary brackets:

```
Map<Integer, List<Employee>> brackets = emps.collect(
    Collectors.groupingBy(
        e->e.getSalary()/1000));
```

We get a map of lists of employees, with key 0 mapping to employees with salary 0-999, 1 to 1000-1999, etc.

To use partitioningBy, we need a predicate.

```
partitioningBy(Predicate<? super T> predicate)
```

This returns a sequential Collector. There is no guarantee on the actual type of map nor list.

Streams of Primitive Types
IntStream, LongStream and DoubleStream - they tend to be more efficient as they avoid the wrapping and unwrapping required for a normal stream. They also add some specialised operations that work with these primitive types.

For example:

```
//Stream<T>
T reduce(T seed, BinaryOperator<T> op)
//IntStream
int reduce(T seed, IntBinaryOperator op)
```

For IntStream:

- IntPredicate: filter, noneMatch, allMatch, anyMatch
- int: of
- IntSupplier: generate
- IntUnaryOperator: iterate
- IntConsumer: forEach, forEachOrdered
- IntUnaryFunction: map
- IntFunction: flatMap

The same names apply for LongStream and DoubleStream.

Specialised Operations

- Range building: range, rangeClosed (on int and long)
- Arithmetics: sum, average
- Others: min, max

```
static IntStream range(int startInclusive, int endExclusive) // excludes end
static IntStream range(int startInclusive, int endInclusive) // includes end
```

Same for LongStream

```
for(int i=0; i<10; i++) do_something(i);
// vs
IntStream.range(0, 10)
         .forEachOrdered(MyClass::do_something);
```

```
OptionalInt sum()
OptionalDouble average() // returns double even for IntStream
OptionalInt min()
OptionalInt max()

//Similarly for LongStream and DoubleStream
```

Mapping from a Generic Stream to a Primitive Stream
This can be done using a method from Stream<T>

```
IntStream mapToInt(ToIntFunction<? super T> mapper)
LongStream mapToLong(ToLongFunction<? super T> mapper)
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)
```

Computing average salary of a stream of employees:

```
OptionalDouble avgSalary = emps.mapToInt(Employee::getSalary).average();
```

Parallel Streams
Two ways to create a parallel stream:

- From Collection
    - Collection::parallelStream
- From any other source
    - Stream::parallel (intermediate op)

They do not change anything about the data, but merely sets a “parallel” stream flag, which tells the machine that any operations on the stream can be run concurrently by different threads.

What if we keep alternating between .parallel() and its counterpart .sequential()? This does not matter, it is the final flag that matters. If the final operation is .parallel(), the whole pipeline is parallel.

In other words, it seems like you have “control” over the operations across different stages, but it is not true. It will all be parallel.

Parallel streams:

- Stream elements are assigned to different threads
- Each thread executes the **whole pipeline** on a **subset** of the elements

Stateless example:

```
// Functionally (sequential)
int totalSalary = emps.stream()
                      .mapToInt(Employee::getSalary)
                      .sum();
```

```
// Functionally (parallel)
int totalSalary = emps.stream()
                      .parallel()
                      .mapToInt(Employee::getSalary)
                      .sum();
```

Stateful example: (NOT GOOD!!)

```
// Imperatively (sequential)
class SalaryAdder{
    int total;
    public void accept(Employee e){
        total += e.getSalary();
    }
}
SalaryAdder adder = new SalaryAdder();

emps.stream().forEach(adder::accept); // method reference acts as a bridge
from the method and its signature to the consumer functional interface
int totalSalary = adder.total;
```

What is the problem?

```
// Imperatively (parallel)
class SalaryAdder{
    int total;
    public void accept(Employee e){
        total += e.getSalary(); // this is a race condition!
    }
}
SalaryAdder adder = new SalaryAdder();

emps.stream().
    .parallel()
    .forEach(adder::accept);

int totalSalary = adder.total;
```

We end up calling the accept method concurrently, and since there is no synchronisation, this will not work and we will end up missing a lot of updates on the total variable.

We need to add synchronisation to this. What is one way? We can do this:

```
public synchronized void accept(Employee e){
    total += e.getSalary(); // this is a race condition!
}
```

This will work, but it will be super slow.

There is one more version. We can actually use a recently added concurrency Adder class. They are specifically optimised for multithreading. It is able to be updated concurrently by multiple threads. We can then:

```
class SalaryAdder{
    LongAdder total;
    public void accept(Employee e){
        total.add(e.getSalary());
    }
}
long totalSalary = adder.total.longValue();
```

This is also correct, BUT it's still substantially slower than the stateless functional way.

Think Functionally
Pillars of Functional-Style Programming:

- Immutable objects
    - All final fields
    - All reference fields point to immutable objects, such as String, Integer etc.
    - Updating an immutable object?
        - You don't!
        - You build a new one.
- Stateless functions
    - Do not modify fields of enclosing object (this)
    - Do not modify static fields of enclosing class or other classes
    - Do not modify the state of its argument
    - Hence, always the same return value for the same arguments
        - Also known as referential transparency
    - A pure function should also have no side effects, so no I/O and no exceptions

Example: List Concatenation

```
MyList<T> a, b;
```

Mutable MyList, stateful concatenation:

```
a.concat(b);
```

Immutable MyList, stateless concatenation:

```
MyList<T> c = MyList.concat(a, b);
```

Wait, then how can an accumulator be stateless? We just did mutable summarisations using an accumulator, which updates the current summary with the next element. How can it be stateless?

It cannot be fully stateless, as it modifies its summary argument. But, given the same summary and same element, it still updates summary in the same way. As such, the java

documentation uses “stateless” in a flexible way.

Using Parallel Streams
Let us look at the parallel performance profile of:

- Data sources
- Intermediate operations
- Terminal operations

and from there derive some guidelines for improving parallel performance.

Firstly, the pipeline of operations can affect the performance of parallelisation, unlike that of sequential, which tends to be more fixed.

- Some operations can parallelise gracefully
- Some resist parallelisation

Here are the critical features to look at for each type of operation:

- Build operations: How easy is it to split the stream?
- Intermediate and terminal operations: Is the operation performed on each element separately, and how easy is it to merge the results?

Build Parallelisation
How is build parallelised? Building parallel streams involves the process of repeated splitting the stream into halves, then handing the halves to a thread each. As such, the build operation’s performance depends on how well the jump to the middle element is supported.

From Collection
This is actually based on complexity:

- Normal array: easy
- ArrayList: easy
- HashSet: medium
- TreeSet: medium
- LinkedList: hard (must traverse first half)

As such, in some cases, it may be useful to first transfer all elements to an array before starting a parallel pipeline.

From Computation
The difficulty is as such:

- generate: easy as each element is generated separately
- IntStream::range: easy as the range is regular and known in advance
- iterate: hard as each element needs to be generated from the previous

Intermediate Operations Parallelisation
Parallel-friendly operations

- filter
- map
- flatMap

They work on each element separately and partial results only need to be concatenated. The condition is that the **functional argument is stateless.**

Parallel-unfriendly operations

- limit
- takeWhile (especially on ordered streams)
- dropWhile (especially on ordered streams)

This is because their result depends on the past, hence we cannot process different chunks of elements separately.

The two of them can be sped-up by unordering the stream, but then they lose much of their meaning and use since they basically end up returning an arbitrary subset of the elements.

More parallel-unfriendly operations

- distinct
- Sort

These operations can process different chunks of elements separately, but the merging process requires partial re-processing. The distinct can be sped-up by unordering the stream.

Terminal Operations Parallelisation
Most are parallel-friendly, provided that the functional arguments are stateless:

- forEach
- count
- allMatch
- anyMatch
- findAny
- findFirst
- reduce
- sum
- max
- min

Standard collectors are parallel-friendly

- toList
- toSet
- etc.

Grouping collectors are relatively efficient

- toMap
- groupingBy
- etc.

If order is not relevant, then it is best to use their concurrent versions:

- toConcurrentMap
- groupingByConcurrent
- etc.

These skip the combining step and operate concurrently on a single summary structure.

For custom mutable summarisations using the collect method:

- Sequential: parallel performance depends on the combiner
    - It is the most expensive operation
- Concurrent: parallel performance depends on the accumulator
    - Since they don't use the combiner

fork-join Framework
This is the framework that supports parallel streams. This is how parallel streams work:

- The stream is split into chunks
- Each thread takes a chunk and processes it
- When threads finish, their results are merged
- The fork-join framework facilitates this entire process

In java.util.concurrent since Java 7.

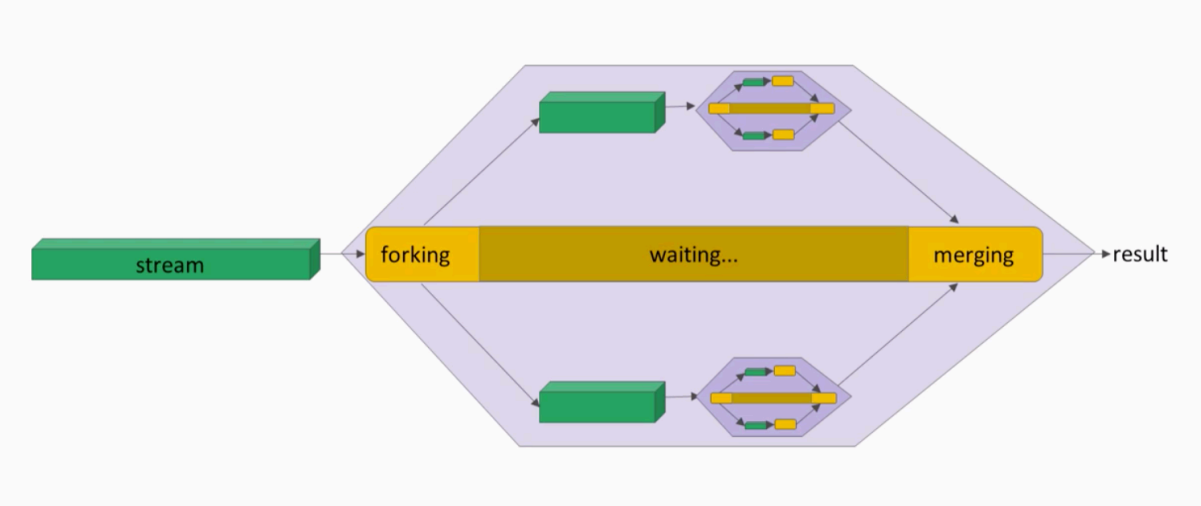
The most important classes are:

- ForkJoinPool - special kind of ExecutorService
- ForkJoinTask - base class for tasks

ExecutorService
Class that accepts tasks and assigns them to a pool of threads. For ForkJoinPool, the number of threads is the number of cores.

ForkJoinPool is a pool of threads that executes tasks. The typical task will:

- Split/Fork itself into subtasks
- Wait for/Join these subtasks
- Merge the result



The tasks are actually recursively split into smaller subtasks until it doesn't make any sense to split them further.

There is thus substantiate overhead due to the large amount of splitting and merging. Hence, parallel execution must be worth the burden:

- When you have many elements in your stream
- The operations to perform on each element are very time-consuming
- Or both of the above