Functional Programming in Java

INT103 Advanced Programming

2022-2

Asst. Prof. Kriengkrai Porkaew, PhD.

A Computer Program

A computer program is **a collection of instructions** that performs a specific task when executed by a computer.

A computer program is usually written in a human-readable form (called **source code**) by a computer programmer in **a programming language**.

A programming language is a formal language, which comprises a set of instructions that produce various kinds of output.

https://en.wikipedia.org/wiki/Computer_program
https://en.wikipedia.org/wiki/Programming_language

Programming Languages

- **Imperative Programming Languages**
  - **Imperative programming** is a programming paradigm that uses statements that change a program's state. An imperative program consists of commands for the computer to perform.
  - **Procedural programming** is a programming paradigm, derived from imperative programming, based upon the concept of the procedure call. Procedures, also known as routines, subroutines, or functions, simply contain a series of computational steps to be carried out.
  - **Structured programming** is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of the structured control flow constructs of selection (if/then/else) and repetition (while and for), block structures, and subroutines.

https://en.wikipedia.org/wiki/Imperative_programming
https://en.wikipedia.org/wiki/Procedural_programming
https://en.wikipedia.org/wiki/Structured_programming

Object-Oriented Programming

- **Imperative > Procedural & Structured programming**
  - A program can be viewed as a collection of **data** that are stored in variables and **functions** that consist of statements that change the data that are stored in variables.
- **Object-Oriented Programming**
  - Object-oriented programming is derived from imperative > structured > procedural programming.
  - data structures and functions on that data structures are combined to form a small unit of program called **object**.
  - Objects that share the same data structures and functions are objects of the same type called **class**.
  - Functions on objects are called **methods**.
    Data structures in objects are called **attributes** or **fields**.
  - **Objects are stored in variables**.
  - In a sense, **OOP focuses more on data** while **functions are just parts of data**.
    Data can be changed dynamically but **functions cannot be changed**.

Functional Programming Language Characteristics

Functions are **First-Class Citizen**, just like **Data**.

Functions and **Data** are **values**.

Values can be
stored in variables/arrays,
passed as parameters,
returned as outputs.

Declarative Approach vs. Imperative Approach

- Imperative > Procedural & Structured programming
  - A program can be viewed as a collection of **data** that are stored in variables and **functions** that consist of statements that change the data that are stored in variables.
- **Functional Programming**
  - Functional programming languages are **declarative**: describing what is needed to be done (declarative); not how to do it (imperative).
  - **Functions are first-class citizen**
    - Functions can be **treated as values**, just like data.
    - Functions can be **stored in variables**, just like data.
    - Functions can be **passed as arguments** to other functions, just like data.
    - Functions can be **returned as results** from other functions, just like data.
    - Data have types and **functions also have types**.
  - A function needs not to have a name if it is defined and assigned to a variable directly. This function is an **anonymous function**. The expression that defines the function is called **lambda expression**.
  - Functions that can receive other functions as parameters or return other functions as results are called **higher-order functions.**

A little bit of History of Functional Programming Languages

- **Lambda Calculus** (1930s – by **Alonzo Church**)
  - Turing complete
- **Lisp** (1958 – by **John McCarthy**)
  - One of the oldest high-level programming languages (after FORTRAN - 1957)
- **Scheme** (1975)
  - Lisp family, lexical scope, tail-call optimization, continuation
- **Haskell** (1990 – named after **Haskell Curry**)
  - Purely functional programming language
- **Clojure** (2007)
  - Lisp family
- There are many more functional programming languages
  - **ML**, **SASL**, …

Why Functional Programming in Java?

- **Functional Programming Concepts**
  - Functional programming languages are **declarative**:
    describing what is needed to be done; not how to do it.
  - Functional programming prefers **immutable, no state, and no side-effect**:
    data should be immutable;
    functions should not access any state outside the functions;
    and **executions of functions** should have no side-effect to external states;
    functions do not change their behaviors: If same input then same output.
  - The basic principle of functional programming is **Function Composition**.
- **Object-oriented programming concepts**
  - OOP, in a sense, are quite the opposite of functional programming concepts.
  - Most objects are mutable and full of states (states = data in the attributes).
  - Methods cause lots of side-effect (side-effect = change data in the attributes).
- **Why Functional Programming in Java?**
  - More readable code, more concise code vs. boilerplate code
  - Easier to go parallel with functional programming:
    "what to do – not how to do", "immutable", "side-effect free"
  - The power of **function composition**.

Lambda Expression

```
x = a -> a + 1; // assigned to a variable
y[0] = a -> a + 1; // stored in an array
z = function(a -> a + 1); // sent as a parameter
return a -> a + 1; // returned as an output
```

Java: Class -> Anonymous Class -> **Lambda Expression**


```
interface Greeting {
    public void greet();
}
```

() -> void

Take no argument and return nothing

```
class Hello implements Greeting {
    @Override
    public void greet() {
        System.out.println("Hello Class.");
    }
}
```

Class

```
public class FunctionalTest {
    public static void main(String[] args) {
        Greeting [] array = new Greeting[3];
        array[0] = helloClass();
        array[1] = helloAnonymous();
        array[2] = helloLambda();
        for (Greeting var : array) {
            var.greet();
        }
    }
```

function invocation

```
private static Greeting helloClass() {
    return new Hello();
}
```

```
private static Greeting helloAnonymous() {
    return new Greeting() {
        @Override
        public void greet() {
            System.out.println("Hello Anonymous.");
        }
    };
}
```

Anonymous Class

Output =>

```
Hello Class.
Hello Anonymous.
Hello Lambda.
```

```
private static Greeting helloLambda() {
    return () -> System.out.println("Hello Lambda.");
}
```

Lambda Expression => anonymous function
concise code

Lambda Expression

Syntax of Lambda Expression

```
[1] x = (String s) -> { return s.length; };
[2] x = (s) -> { return s.length; };
[3] x = s -> { return s.length; };
[4] x = s -> s.length;
```

- Lambda Expression =
  - **[1]** (*LIST_OF_VARIABLES*) **- > {** *LIST_OF_STATEMENTS_ENDED_WITH_RETURN* **}**
  - *LIST_OF_VARIABLES* = *VARIABLE* [, *VARIABLE*] *
  - *VARIABLE* = **DATA_TYPE VARIABLE_NAME**
  - **[2] DATA_TYPE** can be omitted.
  - **[3]** If there is ***only one variable*** in the ***LIST_OF_VARIABLES***, ( ) can be omitted.
  - **[4]** If there is ***only one statement*** in the ***LIST_OF_STATEMENTS***, **{ }** and the keyword **return** in the statement are omitted.

```
x = (int s, int t) -> { return s + t; };
x = (s, t) -> s + t;
```

- **Lambda Expression** is an expression, so it can be …
  - assigned to a variable (waiting to be executed),
  - stored in an array or a collection as a value (waiting to be executed),
  - returned from a function as a value (waiting to be executed),
  - passed to another function as an argument (waiting to be executed).

  It can be executed using the single abstract method in the functional interface.

Syntax of Lambda Expression

```
String substring(String str, int offet, int length) {
    if (offet<0 || length<=0 || offset+length>str.length())
        throw new IllegalArgumentException();
    return str.substring(offset, offset+length);
}
```

Transform a method to a lambda expression

```
(String str, int offet, int length) -> {
    if (offet<0 || length<=0 || offset+length>str.length())
        throw new IllegalArgumentException();
    return str.substring(offset, offset+length);
}
```

If data type is omitted.

```
(str, offet, length) -> {
    if (offet<0 || length<=0 || offset+length>str.length())
        throw new IllegalArgumentException();
    return s.substring(offset, offset+length);
}
```

If arguments are not validated.

```
(str, off, len) -> {
    return s.substring(off, off+len);
}
```

If there is only one statement, **{ }** and **return** can be omitted.

```
(str, off, len) -> str.substring(off, off+len);
```

Examples of Lambda Expressions

Since functions/lambdas also need to have a type:
Interfaces act as the types of functions/lambdas.
These interfaces must have **only one abstract method**.
These interfaces are called **functional interface**.

```
public class FunctionalTest2 {
    public static void main(String[] args) {
        Greeting1 g1;
        Greeting2 g2;
        Greeting3 g3, g4;
        g1 = s -> System.out.println("G1: Hello, " + s);
        g2 = (s, m) -> System.out.println("G2: " + m + ", " + s);
        g3 = s -> "G3: Hello, " + s;
        g4 = s -> {
                String cap = s.toUpperCase();
                return "G4: Hello, " + cap;
             };

        g1.greet("you");
        g2.greet("Lambda", "Good Day");
        System.out.println(g3.greet("Simple Lambda"));
        System.out.println(g4.greet("A Little Complex Lambda"));
    }
}
```

Lambda Expressions

```
interface Greeting1 {
    public void greet(String someone);
}
```
String -> void

```
interface Greeting2 {
    public void greet(String someone, String message);
}
```
(String, String) -> void

```
interface Greeting3 {
    public String greet(String someone);
}
```
String -> String

Output =>

```
G1: Hello, you
G2: Good Day, Lambda
G3: Hello, Simple Lambda
G4: Hello, A LITTLE COMPLEX LAMBDA
```

Another Example of Lambda Expression in Java

```java
public class OOCalculator implements Calculator {
    double left, right;
    String operator;

    public OOCalculator(double left, double right, String symbol) {
        this.left = left; this.right = right;
        this.operator = operation(symbol);
    }

    private String operation(String symbol) {
        switch (symbol) {
            case "+": case "*": case "-": case "/":
            case "%": return symbol;
            default: throw new IllegalArgumentException();
        }
    }

    @Override
    public double compute() {
        switch (operator) {
            case "+": return left + right;
            case "*": return left * right;
            case "-": return left - right;
            case "/": return left / right;
            case "%": return left % right;
            default: throw new IllegalStateException();
        }
    }
}
```

```java
@FunctionalInterface
public interface DoubleBinaryOperator {
    double applyAsDouble(double left, double right);
}
```

(double, double) -> double

```java
public class FunctionalCalculator implements Calculator {
    double left, right;
    DoubleBinaryOperator operator;

    public FunctionalCalculator(double left, double right,
            String symbol) {
        this.left = left; this.right = right;
        this.operator = operation(symbol);
    }

    private DoubleBinaryOperator operation(String symbol) {
        switch (symbol) {
            case "+": return (d1, d2) -> d1 + d2;
            case "-": return (d1, d2) -> d1 - d2;
            case "*": return (d1, d2) -> d1 * d2;
            case "/": return (d1, d2) -> d1 / d2;
            case "%": return (d1, d2) -> d1 % d2;
            default: throw new IllegalArgumentException();
        }
    }

    @Override
    public double compute() {
        return operator.applyAsDouble(left, right);
    }
}
```

A function is stored in a variable

Functions (lambda expressions) are returned as results

The function is executed (the Java way)

Interfaces/Classes are the **types** for **objects**.

There are no **objects** that are not associated with any class or interface.

(Functional) Interfaces are the **types** for **lambda expressions**.

There are no **lambda expressions** that are not associated with any interface.

Method References

Use **method references** in place of **lambda expressions**.

- Static methods
- Instance methods (for a specific object or for any object)
- Object Constructors
- Array Constructors

Method References

(T t) -> t.method(?) == T::method

static method	→ ClassName::method	→ String::compare
instance method	→ ClassName::method (any)	→ String::compareTo
instance method	→ object::method (specific)	→ str::compareTo
constructor	→ ClassName::new	→ String::new
array constructor	→ ClassName[]::new	→ String[]::new

Lambda Expressions		Method References	Type
(String s0, String s1) -> String.compare(s0, s1)	=	String::compare	ToIntBiFunction
(String s) -> this.compareTo(s)	=	String::compareTo	ToIntFunction
(String s) -> this.compareTo(s)	=	str::compareTo	ToIntFunction
param -> new Student(param)	=	Student::new	IntFunction
n -> new Student[n]	=	Student[]::new	IntFunction

Functional Interfaces in java.util.function

```
@FunctionalInterface
interface InterfaceName {
    ReturnType methodName(ParameterType parameterName);
}
```

Functional Programming in Java

- Java is a **strongly typed** programming language
  - Since functions can be treated as values, **functions need to have types**.
  - Types of functions, in a sense, are **function signatures**.
  - E.g.,
    - a function that receives no argument and returns a boolean-value result have the following type: **() -> boolean** (called **BooleanSupplier**)
    - A function that receives two double-value arguments and returns a double-value result have the following type: **(double, double) -> double** (called **DoubleBinaryOperator**)
  - Package: **java.util.function**
    - Provides 40+ types of functions in the form of interface; each of which is annotated with **@FunctionalInterface**
    - Interfaces that are annotated with @FunctionalInterface are interfaces that **can have only one abstract, non-static, non-default method** and the method signature is the function signature.
    - 40+ types of functions can be grouped into 3 types:
      - **Suppliers**: **receive nothing** as an input but **return** an output value.
      - **Consumers**: **receive** one or two input arguments but **return nothing** as output.
      - **Functions**: **receive** one or two input arguments and **return** an output value.
    - Developers may create their own types of functions anytime, just like creating their own classes.

Predicates
Operators

43 @FunctionalInterface in java.util.function package

No Input Argument

Supplier : get()

Supplier<T> : () -> T
BooleanSupplier : () -> boolean
IntSupplier : () -> int
LongSupplier : () -> long
DoubleSupplier : () -> double

One Input Argument (**Unary**)

Function: Predicate, Operator, Function

(Bi)Function : apply()

Function<T,R> : T -> R
IntFunction<R> : int -> R
LongFunction<R> : long -> R
DoubleFunction<R> : double -> R
ToIntFunction<T> : T -> int
ToLongFunction<T> : T -> long
ToDoubleFunction<T> : T -> double
IntToLongFunction<T> : int -> long
IntToDoubleFunction<T> : int -> double
LongToIntFunction<T> : long -> int
LongToDoubleFunction<T>: long -> double
DoubleToIntFunction<T> : double -> int
DoubleToLongFunction<T>: double -> long

andThen(...)

and(), **or**(...),
negate(), **isEqual**(...)

No Returned Value

(Bi)Consumer : accept()

Return boolean

(Bi)Predicate : test()

Input/Output : Same Type

(Unary/Binary)Operator

Consumer<T> : T -> void
IntConsumer : int -> void
LongConsumer : long -> void
DoubleConsumer : double -> void

Predicate<T> : T -> boolean
IntPredicate : int -> boolean
LongPredicate : long -> boolean
DoublePredicate : double -> boolean

UnaryOperator<T> : T -> T
IntUnaryOperator : int -> int
LongUnaryOperator : long -> long
DoubleUnaryOperator : double -> double

BiConsumer<T,U> : T, U -> void
ObjIntConsumer<T> : T, int -> void
ObjLongConsumer<T> : T, long -> void
ObjDoubleConsumer<T> : T, double -> void

BiPredicate<T,U> : T, U -> boolean

BinaryOperator<T> : T, T -> T
IntBinaryOperator : int, int -> int
LongBinaryOperator : long, long -> long
DoubleBinaryOperator: double, double -> double

BiFunction<T,U,R> : T, U -> R
ToIntBiFunction<T,U> : T, U -> int
ToLongBiFunction<T,U> : T, U -> long
ToDoubleBiFunction<T,U> : T, U -> double

Two Input Arguments (**Binary**)

Other @FunctionalInterface elsewhere; e.g.,

Runnable : run() : () -> void

Callable<V> : call() : () -> V

Comparator<T> : compare() : T, T -> int

Tools/Utilities useful for Functional

Optional & Unmodifiable Collections

- Use **Optional** instead of **null** in some use cases
- **Collections** that **cannot add, remove, or change** the elements but **the elements** in the collection **can still be modified**.

Optional<T>, OptionalInt, OptionalLong, OptionalDouble

Accessing the value

```
boolean      isEmpty()
boolean      isPresent()
void         ifPresent(Consumer<? super T> action)
void         ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)
T            get(); getAsInt/Long/Double() [ use orElse(…), orElseThrow() instead ]
T            orElse(T other)
T            orElseThrow()
Optional<T>  or(Supplier<? extends Optional<? extends T>> supplier)
T            orElseGet(Supplier<? extends T> supplier)
T            orElseThrow(Supplier<? extends X> exceptionSupplier)
```

Object operations

```
boolean  equals(Object obj)
int      hashCode()
String   toString()
```

Stream operations

```
Stream<T>    stream()
Optional<T>  filter(Predicate<? super T> predicate)
Optional<U>  map(Function<? super T,? extends U> mapper)
Optional<U>  flatMap(Function<? super T,? extends Optional<? extends U>> mapper)
```

Create (static)

```
Optional<T>  empty()
Optional<T>  of(T notNullvalue)
Optional<T>  ofNullable(T value)
```

Use **Optional** instead of **null** in some use cases

Unmodifiable Collection/List/Set/Map

```
Collection<T>          unmodifiableCollection(Collection<? extends T> collection)
List<T>                unmodifiableList(List<? extends T> list)
Set<T>                 unmodifiableSet(Set<? extends T> set)
Map<K,V>               unmodifiableMap(Map<? extends K,? extends V> map)
SortedSet<T>           unmodifiableSortedSet(SortedSet<T> set)
SortedMap<K,V>         unmodifiableSortedMap(SortedMap<K,? extends V> map)
NavigableSet<T>        unmodifiableNavigableSet(NavigableSet<T> set)
NavigableMap<K,V>      unmodifiableNavigableMap(NavigableMap<K,? extends V> map)
```

Unmodifiable View of Collections

- **cannot add** new elements
- **cannot remove** old elements
- **cannot swap** elements
- **cannot replace** elements

But **elements can still be modified**.

```
List<E>      List.of(E … values)
List<E>      List.copyOf(Collection<E> collection)
Set<E>       Set.of(E … values)
Set<E>       Set.copyOf(Collection<E> collection)
Map<K,V>     Map.of(K k, V v, …)
Map<K,V>     Map.ofEntries(Entry<K,V> … entries)
Map<K,V>     Map.copyOf(Map<K,V> map)
```

Java Stream API
for Functional Programming

Stream Creation: Stream.of(), collection.stream()

Stream Non-Terminal Operations: filter(), map(), sort(), skip(), limit(), …

Stream Terminal Operations: reduce(), collect(), forEach(), aggregation (count, …)

Java Stream API

Implementing Higher-Order Functions

- filter()
- map()
- sorted()
- …
- collect()
- reduce()
- forEach()
- …
- count()
- min()
- max()
- …

Hierarchy For Package java.util.stream

Package Hierarchies:
All Packages

https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

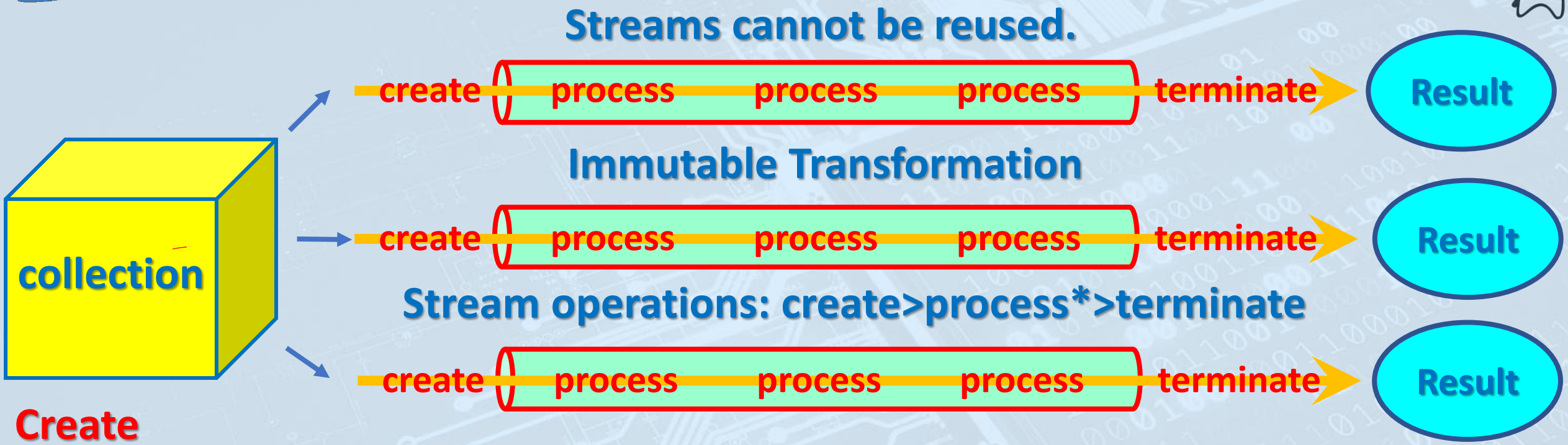
Class Hierarchy

- java.lang.**Object**
  - java.util.stream.**Collectors**
  - java.util.stream.**StreamSupport**

Interface Hierarchy

- java.lang.**AutoCloseable**
  - java.util.stream.**BaseStream**<T,S>
    - java.util.stream.**DoubleStream**
    - java.util.stream.**IntStream**
    - java.util.stream.**LongStream**
    - java.util.stream.**Stream**<T>
- java.util.stream.**Collector**<T,A,R>
- java.util.function.**Consumer**<T>
  - java.util.stream.**Stream.Builder**<T>
- java.util.function.**DoubleConsumer**
  - java.util.stream.**DoubleStream.Builder**
- java.util.function.**IntConsumer**
  - java.util.stream.**IntStream.Builder**
- java.util.function.**LongConsumer**
  - java.util.stream.**LongStream.Builder**

Java Stream API



Create

Stream<T>.**of**(T … **values**)

Stream<T>.**of**(T[] **values**)

collection<T>.**stream**()

collection<T>.**parallelStream**()

StreamSupport.**stream**(**spliterator<T>**, **parallelBoolean**)

StreamSupport.**stream**(**iterable<T>.spliterator**(), **parallelBoolean**)

Java Stream API

java.util.stream

Interfaces

- BaseStream<T, S>
  - Stream<T>
  - IntStream
  - LongStream
  - DoubleStream
- Collector

Classes

- Collectors
- StreamSupport

BaseStream<T, S>

- parallel()
- sequential()
- unordered()
- iterator() // terminal

Stream<T> static operations

Create

static Stream<T> **concat**(Stream<? extends T> **s0**, Stream<? extends T> **s1**)
static Stream<T> **generate**(Supplier<T> **supply**)
static Stream<T> **iterate**(T **seed**, UnaryOperator<T> **func**)
static Stream<T> **iterate**(T **seed**, Predicate<? super T> **hasNext**, UnaryOperator<T> **next**)
static Stream<T> **empty**(), **of**(T... values), **of**(T t), **ofNullable**(T t)

collection.stream()
collection.parallelStream()
Arrays.**stream**(**array**)

Stream<T> non-terminal operations

Process

Stream<T> **skip**(long **n**), **limit**(long **maxSize**), **distinct**()
Stream<T> **sorted**(), **sorted**(Comparator<? super T> **comparator**)
Stream<T> **filter**(Predicate<? super T> **predicate**)
Stream<T> **peek**(Consumer<? super T> **action**)
Stream<T> **takeWhile/dropWhile**(Predicate<? super T> **predicate**)

Stream<T> -> Another Stream

Stream<R> **map**(Function<? super T,? extends R> **mapper**)
Stream<R> **mapMulti**(BiConsumer<? super T,? super Consumer<R>> **mapper**)
Stream<R> **flatMap**(Function<? super T,? extends Stream<? extends R>> **mapper**)
IntStream **mapToInt**(ToIntFunction<? super T> **mapper**) + Long|Double
IntStream **mapMultiToInt**(BiConsumer<? super T,? super IntConsumer> **mapper**)
IntStream **flatMapToInt**(Function<? super T,? extends IntStream> **mapper**)

Stream<T> Terminal Operations

Terminate

boolean operations

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

Statistics

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

Reduction

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

Retrieving One

```
Optional<T>  findAny()
Optional<T>  findFirst()
```

To Collection

```
List<T>   toList()
Object[]  toArray()
A[]       toArray(IntFunction<A[]> generator)
```

ForEach

```
void  forEach(Consumer<? super T> action)
void  forEachOrdered(Consumer<? super T> action)
```

Collecting

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

[Int|Long|Double]Stream operations (+)

Create

static IntStream	**range**(int **startInclusive**, int **endExclusive**)
static IntStream	**rangeClosed**(int **startInclusive**, int **endinclusive**)

Process

IntStream	**mapToInt**(**&**ToIntFunction **mapper**)
LongStream	**mapToLong**(**&**ToLongFunction **mapper**)
DoubleStream	**mapToDouble**(**&**ToDoubleFunction **mapper**)
Stream<U>	**mapToObj**(**&**Function<? extends U> **mapper**)
Stream<**#**>	**boxed**()
LongStream	**asLongStream**()
DoubleStream	**asDoubleStream**()

Terminate

$	**sum**()
OptionalDouble	**average**()
&SummaryStatistics	**summaryStatistics**()
$[]	**toArray**()

& = **Int** | **Long** | **Double**
$ = **int** | **long** | **double**
# = **Integer** | **Long** | **Double**

Tools/Utilities useful for Streams

StreamSupport & Collector

StreamSupport Stream Creation from **Spliterator**

Collector Constructor

supplier	storing the result
accumulator	accumulating elements
combiner	combining the results from parallel processing
finisher	finishing the result
characteristics	CONCURRENT, IDENTITY_FINISH, UNORDERED

StreamSupport & Collector

Stream Creation from Spliterator

StreamSupport (static methods)

```
Stream<T>       stream(
                    Spliterator<T> spliterator,
                    boolean parallel)
IntStream       intStream(
                    Spliterator.OfInt spliterator,
                    boolean parallel)
LongStream      longStream(
                    Spliterator.OfLong spliterator,
                    boolean parallel)
DoubleStream    doubleStream(
                    Spliterator.OfDouble spliterator,
                    boolean parallel)
```

Collector (static methods)

```
Collector<T,A,R>   of(
        Supplier<A> supplier,
        BiConsumer<A,T> accumulator,
        BinaryOperator<A> combiner,
        Function<A,R> finisher,
        Collector.Characteristics... characteristics)
Collector<T,R,R>   of(
        Supplier<R> supplier,
        BiConsumer<R,T> accumulator,
        BinaryOperator<R> combiner,
        Collector.Characteristics... characteristics)
```

Collector.Characteristics enum

- CONCURRENT
- IDENTITY_FINISH
- UNORDERED

Utilities for Collector

Collectors : Collector Creators

Pre-Processing : filter/map/flatMap().collect()

Post-Processing : collect().finish(), collect().merge()

Processing : reduce()

Aggregation : count(), min(), max(), sum(), average(), stats(), join(); groupBy()

To Collections : toCollection(), toList(), toSet(), toMap()

Collectors (static methods)

```
Collector<T,?,R>   teeing(
                       Collector<? super T,?,R1> downstream1,
                       Collector<? super T,?,R2> downstream2,
                       BiFunction<? super R1,? super R2,R> merger)
```

filter().collect()
map().collect()
flatMap().collect()
reduce()
collect().finish()
tee().merge()

```
Collector<T,?,Optional<T>>   reducing(
                                 BinaryOperator<T> op)
Collector<T,?,T>             reducing(T identity,
                                 BinaryOperator<T> op)
Collector<T,?,U>             reducing(U identity,
                                 Function<? super T,? extends U> mapper,
                                 BinaryOperator<U> op)
```

```
Collector<T,?,R>  filtering(
                      Predicate<? super T> predicate,
                      Collector<? super T,A,R> downstream)
Collector<T,?,R>  mapping(
                      Function<? super T,? extends U> mapper,
                      Collector<? super U,A,R> downstream)
Collector<T,?,R>  flatMapping(
                      Function<? super T,? extends Stream<? extends U>> mapper,
                      Collector<? super U,A,R> downstream)
```

```
Collector<T,A,RR>  collectingAndThen(
                       Collector<T,A,R> downstream,
                       Function<R,RR> finisher)
```

Collectors (static methods)

Collector<T,?,Long> **counting**()

Object Count

Collector<T,?,Optional<T>> **minBy**(Comparator<? super T> **comparator**)
Collector<T,?,Optional<T>> **maxBy**(Comparator<? super T> **comparator**)

Comparable Min Max

Collector<T,?,Integer> **summingInt**(ToIntFunction<? super T> **mapper**)
Collector<T,?,Long> **summingLong**(ToLongFunction<? super T> **mapper**)
Collector<T,?,Double> **summingDouble**(ToDoubleFunction<? super T> **mapper**)

Collector<T,?,Double> **averagingInt**(ToIntFunction<? super T> **mapper**)
Collector<T,?,Double> **averagingLong**(ToLongFunction<? super T> **mapper**)
Collector<T,?,Double> **averagingDouble**(ToDoubleFunction<? super T> **mapper**)

Collector<T,?,IntSummaryStatistics> **summarizingInt**(ToIntFunction<? super T> **mapper**)
Collector<T,?,LongSummaryStatistics> **summarizingLong**(ToLongFunction<? super T> **mapper**)
Collector<T,?,DoubleSummaryStatistics> **summarizingDouble**(ToDoubleFunction<? super T> **mapper**)

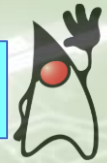
Object -> number
Sum Average Stat.

count()
---
min(),max()
---
sum(),average(),
statistics()
---
join()

Collector<CharSequence,?,String> **joining**()
Collector<CharSequence,?,String> **joining**(CharSequence **delimiter**)
Collector<CharSequence,?,String> **joining**(CharSequence **delimiter**, CharSequence **prefix**, CharSequence **suffix**)

String Join

Collectors (static methods)



To Set, List

```
Collector<T,?,Set<T>>   toSet()
Collector<T,?,List<T>>  toList()
Collector<T,?,Set<T>>   toUnmodifiableSet()
Collector<T,?,List<T>>  toUnmodifiableList()
```

To any collection

```
Collector<T,?,C>toCollection(Supplier<C> collectionFactory)
```

toCollection()
toSet()
toList()
toMap()

To Map

```
[1] Collector<T,?,Map<K,U>>   toMap(
                                Function<? super T,? extends K> keyMapper,
                                Function<? super T,? extends U> valueMapper)
[2] Collector<T,?,Map<K,U>>   toMap(
                                Function<? super T,? extends K> keyMapper,
                                Function<? super T,? extends U> valueMapper,
                                BinaryOperator<U> mergeFunction)
[3] Collector<T,?,M>          toMap(
                                Function<? super T,? extends K> keyMapper,
                                Function<? super T,? extends U> valueMapper,
                                BinaryOperator<U> mergeFunction,
                                Supplier<M> mapFactory)
```

To Map (cont.)

toUnmodifiableMap(…) : SAME AS **toMap**(…) **[1,2]**
toConcurrentMap(…) : SAME AS **toMap**(…) → **ConcurrentMap**<K,U>> : **Map**<K,U>

Collectors (static methods)

groupBy().toMap()

Grouping to Map

```
Collector<T,?,Map<K,List<T>>>  groupingBy(
                                   Function<? super T,? extends K> classifier)
Collector<T,?,Map<K,D>>        groupingBy(
                                   Function<? super T,? extends K> classifier,
                                   Collector<? super T,A,D> downstream)
Collector<T,?,M>               groupingBy(
                                   Function<? super T,? extends K> classifier,
                                   Supplier<M> mapFactory,
                                   Collector<? super T,A,D> downstream)
```

groupingByConcurrent(…) : SAME AS groupingBy(…) → ConcurrentMap<K,U>> : Map<K,U>

True/False Grouping to Map

```
Collector<T,?,Map<Boolean,List<T>>>  partitioningBy(
                                         Predicate<? super T> predicate)
Collector<T,?,Map<Boolean,D>>        partitioningBy(
                                         Predicate<? super T> predicate,
                                         Collector<? super T,A,D> downstream)
```