Java 8 stream:

1. [Java 8 Stream](https://www.journaldev.com/2774/java-8-stream#stream-overview)
2. [Collections and Java Stream](https://www.journaldev.com/2774/java-8-stream#stream-collections)
3. [Functional Interfaces in Java 8 Stream](https://www.journaldev.com/2774/java-8-stream#functional-interfaces)
   1. [Function and BiFunction](https://www.journaldev.com/2774/java-8-stream#function-bifunction)
   2. [Predicate and BiPredicate](https://www.journaldev.com/2774/java-8-stream#predicate-bipredicate)
   3. [Consumer and BiConsumer](https://www.journaldev.com/2774/java-8-stream#consumer-biconsumer)
   4. [Supplier](https://www.journaldev.com/2774/java-8-stream#supplier)
4. [java.util.Optional](https://www.journaldev.com/2774/java-8-stream#java.util.optional)
5. [java.util.Spliterator](https://www.journaldev.com/2774/java-8-stream#java.util.spliterator)
6. [Java Stream Intermediate and Terminal Operations](https://www.journaldev.com/2774/java-8-stream#stream-intermediate-terminal-operations)
7. [Java Stream Short Circuiting Operations](https://www.journaldev.com/2774/java-8-stream#stream-short-circuiting-operations)
8. [Java Stream Examples](https://www.journaldev.com/2774/java-8-stream#java-stream-examples)
   1. [Creating Java Streams](https://www.journaldev.com/2774/java-8-stream#create-stream)
   2. [Converting Java Stream to Collection or Array](https://www.journaldev.com/2774/java-8-stream#stream-to-collection-array)
   3. [Java Stream Intermediate Operations](https://www.journaldev.com/2774/java-8-stream#stream-intermediate-operations)
   4. [Java Stream Terminal Operations](https://www.journaldev.com/2774/java-8-stream#stream-terminal-operations)
9. [Java 8 Stream API Limitations](https://www.journaldev.com/2774/java-8-stream#stream-api-limitations)
10. **Java Stream**
11. Before we look into Java Stream API Examples, let’s see why it was required. Suppose we want to iterate over a list of integers and find out sum of all the integers greater than 10.
12. Prior to Java 8, the approach to do it would be:
13. private static int sumIterator(List<Integer> list) {
14. Iterator<Integer> it = list.iterator();
15. int sum = 0;
16. while (it.hasNext()) {
17. int num = it.next();
18. if (num > 10) {
19. sum += num;
20. }
21. }
22. return sum;
23. }
24. There are three major problems with the above approach:

We just want to know the sum of integers but we would also have to provide how the iteration will take place, this is also called **external iteration** because client program is handling the algorithm to iterate over the list.

D pgm is sequtial in nature, der is no way v can do this in pllel easily. der is a lot of code to do even a siple task.

To overcome all the above shortcomings, Java 8 Stream API was introduced. We can use Java Stream API to implement **internal iteration**, that is better because java framework is in control of the iteration.

**Internal iteration** provides several features such as sequential and parallel execution, filtering based on the given criteria, mapping etc.

Most of the Java 8 Stream API method arguments are functional interfaces, so lambda expressions work very well with them. Let’s see how can we write above logic in a single line statement using Java Streams.

private static int sumStream(List<Integer> list) {

return list.stream().filter(i -> i > 10).mapToInt(i -> i).sum();

}

Notice that abv pgm utilizes java f/w iteration strategy, filtering & mapping methods and would increase efficiency.First of all we will look into the core concepts of Java 8 Stream API and then we will go through some examples for understanding most commonly used methods.

**Collections and Java Stream**

A collection is an in-memory data structure to hold values and before we start using collection, all the values should have been populated. Whereas a java Stream is a data structure that is computed on-demand.

Java Stream doesn’t store data, it operates on d source DS (collection and array) & produce pipelined data that v can use and perform specific opertions. Such as v can create a stream 4m d list & filter it based on a condition.

Java Stream operations use functional interfaces, that makes it a very good fit for functional programming using lambda expression. As you can see in the above example that using lambda expressions make our code readable and short.

Java 8 Stream internal iteration principle helps in achieving lazy-seeking in some of the stream operations. For example filtering, mapping, or duplicate removal can be implemented lazily, allowing higher performance and scope for optimization.

Java Streams are consumable, so there is no way to create a reference to stream for future usage. Since the data is on-demand, it’s not possible to reuse the same stream multiple times.

Java 8 Stream support sequential as well as parallel processing, parallel processing can be very helpful in achieving high performance for large collections.

All the Java Stream API interfaces and classes are in the java.util.stream package. Since we can use primitive data types such as int, long in the collections using auto-boxing and these operations could take a lot of time, there are specific classes for primitive types – IntStream, LongStream and DoubleStream.

**Functional Interfaces in Java 8 Stream**

Some of the commonly used functional interfaces in the Java 8 Stream API methods are:

1. **Function and BiFunction**: Function represents a function that takes one type of argument and returns another type of argument. Function<T, R> is the generic form where T is the type of the input to the function and R is the type of the result of the function.

For handling primitive types, there are specific Function interfaces – ToIntFunction, ToLongFunction, ToDoubleFunction, ToIntBiFunction, ToLongBiFunction, ToDoubleBiFunction, LongToIntFunction, LongToDoubleFunction, IntToLongFunction, IntToDoubleFunction etc.

Some of the Stream methods where Function or it’s primitive specialization is used are:

* + <R> Stream<R> map(Function<? super T, ? extends R> mapper)
  + IntStream mapToInt(ToIntFunction<? super T> mapper) – similarly for long and double returning primitive specific stream.
  + IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper) – similarly for long and double
  + <A> A[] toArray(IntFunction<A[]> generator)
  + <U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)

1. **Predicate and BiPredicate**: It represents a predicate against which elements of the stream are tested. This is used to filter elements from the java stream. Just like Function, there are primitive specific interfaces for int, long and double.

Some of the Stream methods where Predicate or BiPredicate specializations are used are:

* + Stream<T> filter(Predicate<? super T> predicate)
  + boolean anyMatch(Predicate<? super T> predicate)
  + boolean allMatch(Predicate<? super T> predicate)
  + boolean noneMatch(Predicate<? super T> predicate)

1. **Consumer and BiConsumer**: It represents an operation that accepts a single input argument and returns no result. It can be used to perform some action on all the elements of the java stream.

Some of the Java 8 Stream methods where Consumer, BiConsumer or it’s primitive specialization interfaces are used are:

* + Stream<T> peek(Consumer<? super T> action)
  + void forEach(Consumer<? super T> action)
  + void forEachOrdered(Consumer<? super T> action)

1. **Supplier**: Supplier represent an operation through which we can generate new values in the stream. Some of the methods in Stream that takes Supplier argument are:
   * public static<T> Stream<T> generate(Supplier<T> s)
   * <R> R collect(Supplier<R> supplier,BiConsumer<R, ? super T> accumulator,BiConsumer<R, R> combiner)

**java.util.Optional**

Java Optional is a container object which may or may not contain a non-null value. If a value is present, isPresent() will return true and get() will return the value. Stream terminal operations return Optional object. Some of these methods are:

* Optional<T> reduce(BinaryOperator<T> accumulator)
* Optional<T> min(Comparator<? super T> comparator)
* Optional<T> max(Comparator<? super T> comparator)
* Optional<T> findFirst()
* Optional<T> findAny()

**java.util.Spliterator**

For supporting parallel execution in Java 8 Stream API, Spliterator interface is used. Spliterator trySplitmethod returns a new Spliterator that manages a subset of the elements of the original Spliterator.

**Java Stream Intermediate and Terminal Operations**

Java Stream API operations that returns a new Stream are called intermediate operations. Most of the times, these operations are lazy in nature, so they start producing new stream elements and send it to the next operation. Intermediate operations are never the final result producing operations. Commonly used intermediate operations are filter and map.

Java 8 Stream API operations that returns a result or produce a side effect. Once the terminal method is called on a stream, it consumes the stream and after that we can’t use stream. Terminal operations are eager in nature i.e they process all the elements in the stream before returning the result. Commonly used terminal methods are forEach, toArray, min, max, findFirst, anyMatch, allMatch etc. You can identify terminal methods from the return type, they will never return a Stream.

### Java Stream Short Circuiting Operations

An intermediate operation is called short circuiting, if it may produce finite stream for an infinite stream. For example limit() and skip() are two short circuiting intermediate operations.

A terminal operation is called short circuiting, if it may terminate in finite time for infinite stream. For example anyMatch, allMatch, noneMatch, findFirst and findAny are short circuiting terminal operations.

### Java Stream Examples

I have covered almost all the important parts of the Java 8 Stream API. It’s exciting to use this new API features and let’s see it in action with some java stream examples.

#### Creating Java Streams

There are several ways through which we can create a java stream from array and collections. Let’s look into these with simple examples.

1. We can use Stream.of() to create a stream from similar type of data. For example, we can create Java Stream of integers from a group of int or Integer objects.

Stream<Integer> stream = Stream.of(1,2,3,4);

1. We can use Stream.of() with an array of Objects to return the stream. Note that it doesn’t support autoboxing, so we can’t pass primitive type array.
2. Stream<Integer> stream = Stream.of(new Integer[]{1,2,3,4});
3. //works fine
4. Stream<Integer> stream1 = Stream.of(new int[]{1,2,3,4});

//Compile time error, Type mismatch: cannot convert from Stream<int[]> to Stream<Integer>

1. We can use Collection stream() to create sequential stream and parallelStream() to create parallel stream.
2. List<Integer> myList = new ArrayList<>();
3. for(int i=0; i<100; i++) myList.add(i);
5. //sequential stream
6. Stream<Integer> sequentialStream = myList.stream();
8. //parallel stream

Stream<Integer> parallelStream = myList.parallelStream();

1. We can use Stream.generate() and Stream.iterate() methods to create Stream.
2. Stream<String> stream1 = Stream.generate(() -> {return "abc";});

Stream<String> stream2 = Stream.iterate("abc", (i) -> i);

1. Using Arrays.stream() and String.chars() methods.
2. LongStream is = Arrays.stream(new long[]{1,2,3,4});

IntStream is2 = "abc".chars();

#### Converting Java Stream to Collection or Array

There are several ways through which we can get a Collection or Array from a java Stream.

1. We can use java Stream collect() method to get List, Map or Set from stream.
2. Stream<Integer> intStream = Stream.of(1,2,3,4);
3. List<Integer> intList = intStream.collect(Collectors.toList());
4. System.out.println(intList); //prints [1, 2, 3, 4]
5. intStream = Stream.of(1,2,3,4); //stream is closed, so we need to create it again
6. Map<Integer,Integer> intMap = intStream.collect(Collectors.toMap(i -> i, i -> i+10));

System.out.println(intMap); //prints {1=11, 2=12, 3=13, 4=14}

1. We can use stream toArray() method to create an array from the stream.
2. Stream<Integer> intStream = Stream.of(1,2,3,4);
3. Integer[] intArray = intStream.toArray(Integer[]::new);

System.out.println(Arrays.toString(intArray)); //prints [1, 2, 3, 4]

### Java Stream Intermediate Operations

Let’s look into commonly used java Stream intermediate operations example.

1. **Stream filter() example**: We can use filter() method to test stream elements for a condition and generate filtered list.
2. List<Integer> myList = new ArrayList<>();
3. for(int i=0; i<100; i++) myList.add(i);
4. Stream<Integer> sequentialStream = myList.stream();
5. Stream<Integer> highNums = sequentialStream.filter(p -> p > 90); //filter numbers greater than 90
6. System.out.print("High Nums greater than 90=");
7. highNums.forEach(p -> System.out.print(p+" "));

//prints "High Nums greater than 90=91 92 93 94 95 96 97 98 99 "

1. **Stream map() example**: We can use map() to apply functions to an stream. Let’s see how we can use it to apply upper case function to a list of Strings.
2. Stream<String> names = Stream.of("aBc", "d", "ef");
3. System.out.println(names.map(s -> {
4. return s.toUpperCase();
5. }).collect(Collectors.toList()));

//prints [ABC, D, EF]

1. **Stream sorted() example**: We can use sorted() to sort the stream elements by passing Comparator argument.
2. Stream<String> names2 = Stream.of("aBc", "d", "ef", "123456");
3. List<String> reverseSorted = names2.sorted(Comparator.reverseOrder()).collect(Collectors.toList());
4. System.out.println(reverseSorted); // [ef, d, aBc, 123456]
5. Stream<String> names3 = Stream.of("aBc", "d", "ef", "123456");
6. List<String> naturalSorted = names3.sorted().collect(Collectors.toList());

System.out.println(naturalSorted); //[123456, aBc, d, ef]

1. **Stream flatMap() example**: We can use flatMap() to create a stream from the stream of list. Let’s see a simple example to clear this doubt.
2. Stream<List<String>> namesOriginalList = Stream.of(
3. Arrays.asList("Pankaj"),
4. Arrays.asList("David", "Lisa"),
5. Arrays.asList("Amit"));
6. //flat the stream from List<String> to String stream
7. Stream<String> flatStream = namesOriginalList
8. .flatMap(strList -> strList.stream());

flatStream.forEach(System.out::println);

### Java Stream Terminal Operations

Let’s look at some of the java stream terminal operations example.

1. **Stream reduce() example**: We can use reduce() to perform a reduction on the elements of the stream, using an associative accumulation function, and return an Optional. Let’s see how we can use it multiply the integers in a stream.
2. Stream<Integer> numbers = Stream.of(1,2,3,4,5);
4. Optional<Integer> intOptional = numbers.reduce((i,j) -> {return i\*j;});

if(intOptional.isPresent()) System.out.println("Multiplication = "+intOptional.get()); //120

1. **Stream count() example**: We can use this terminal operation to count the number of items in the stream.
2. Stream<Integer> numbers1 = Stream.of(1,2,3,4,5);

System.out.println("Number of elements in stream="+numbers1.count()); //5

1. **Stream forEach() example**: This can be used for iterating over the stream. We can use this in place of iterator. Let’s see how to use it for printing all the elements of the stream.
2. Stream<Integer> numbers2 = Stream.of(1,2,3,4,5);

numbers2.forEach(i -> System.out.print(i+",")); //1,2,3,4,5,

1. **Stream match() examples**: Let’s see some of the examples for matching methods in Stream API.
2. Stream<Integer> numbers3 = Stream.of(1,2,3,4,5);
3. System.out.println("Stream contains 4? "+numbers3.anyMatch(i -> i==4));
4. //Stream contains 4? true
5. Stream<Integer> numbers4 = Stream.of(1,2,3,4,5);
6. System.out.println("Stream contains all elements less than 10? "+numbers4.allMatch(i -> i<10));
7. //Stream contains all elements less than 10? true
8. Stream<Integer> numbers5 = Stream.of(1,2,3,4,5);
9. System.out.println("Stream doesn't contain 10? "+numbers5.noneMatch(i -> i==10));

//Stream doesn't contain 10? true

1. **Stream findFirst() example**: This is a short circuiting terminal operation, let’s see how we can use it to find the first string from a stream starting with D.
2. Stream<String> names4 = Stream.of("Pankaj","Amit","David", "Lisa");
3. Optional<String> firstNameWithD = names4.filter(i -> i.startsWith("D")).findFirst();
4. if(firstNameWithD.isPresent()){
5. System.out.println("First Name starting with D="+firstNameWithD.get()); //David

}

### Java 8 Stream API Limitations

Java 8 Stream API brings a lot of new stuffs to work with list and arrays, but it has some limitations too.

1. **Stateless lambda expressions**: If you are using parallel stream and lambda expressions are stateful, it can result in random responses. Let’s see it with a simple program.

StatefulParallelStream.java

package com.journaldev.java8.stream;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Stream;

public class StatefulParallelStream {

public static void main(String[] args) {

List<Integer> ss = Arrays.asList(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15);

List<Integer> result = new ArrayList<Integer>();

Stream<Integer> stream = ss.parallelStream();

stream.map(s -> {

synchronized (result) {

if (result.size() < 10) {

result.add(s);

}

}

return s;

}).forEach( e -> {});

System.out.println(result);

}

}

If we run above program, you will get different results because it depends on the way stream is getting iterated and we don’t have any order defined for parallel processing. If we use sequential stream, then this problem will not arise.

1. Once a Stream is consumed, it can’t be used later on. As you can see in above examples that every time I am creating a stream.
2. There are a lot of methods in Stream API and the most confusing part is the overloaded methods. It makes the learning curve time taking.

# Java 8 Functional Interfaces

Welcome to the Java 8 functional interfaces example tutorial. Java has always been an **Object Oriented Programming** language. What is means that everything in java programming revolves around Objects (except some primitive types for simplicity). We don’t have only functions in java, they are part of Class and we need to use the class/object to invoke any function.

If we look into some other programming languages such as C++, JavaScript; they are called **functional programming language** because we can write functions and use them when required. Some of these languages support Object Oriented Programming as well as Functional Programming.

Being object oriented is not bad, but it brings a lot of verbosity to the program. For example, let’s say we have to create an instance of Runnable. Usually we do it using anonymous classes like below.

Runnable r = new Runnable(){

@Override

public void run() {

System.out.println("My Runnable");

}};

If you look at the above code, the actual part that is of use is the code inside run() method. Rest all of the code is because of the way java programs are structured.

Java 8 Functional Interfaces and Lambda Expressions help us in writing smaller and cleaner code by removing a lot of boiler-plate code.

### Java 8 Functional Interface

An interface with exactly one abstract method is called Functional Interface. @FunctionalInterfaceannotation is added so that we can mark an interface as functional interface.

It is not mandatory to use it, but it’s best practice to use it with functional interfaces to avoid addition of extra methods accidentally. If the interface is annotated with @FunctionalInterface annotation and we try to have more than one abstract method, it throws compiler error.

The major benefit of java 8 functional interfaces is that we can use **lambda expressions** to instantiate them and avoid using bulky anonymous class implementation.

Java 8 Collections API has been rewritten and new Stream API is introduced that uses a lot of functional interfaces. Java 8 has defined a lot of functional interfaces in java.util.function package. Some of the useful java 8 functional interfaces are Consumer, Supplier, Function and Predicate.

java.lang.Runnable is a great example of functional interface with single abstract method run().

Below code snippet provides some guidance for functional interfaces:

interface Foo { boolean equals(Object obj); }

// Not functional because equals is already an implicit member (Object class)

interface Comparator<T> {

boolean equals(Object obj);

int compare(T o1, T o2);

}

// Functional because Comparator has only one abstract non-Object method

interface Foo {

int m();

Object clone();

}

// Not functional because method Object.clone is not public

interface X { int m(Iterable<String> arg); }

interface Y { int m(Iterable<String> arg); }

interface Z extends X, Y {}

// Functional: two methods, but they have the same signature

interface X { Iterable m(Iterable<String> arg); }

interface Y { Iterable<String> m(Iterable arg); }

interface Z extends X, Y {}

// Functional: Y.m is a subsignature & return-type-substitutable

interface X { int m(Iterable<String> arg); }

interface Y { int m(Iterable<Integer> arg); }

interface Z extends X, Y {}

// Not functional: No method has a subsignature of all abstract methods

interface X { int m(Iterable<String> arg, Class c); }

interface Y { int m(Iterable arg, Class<?> c); }

interface Z extends X, Y {}

// Not functional: No method has a subsignature of all abstract methods

interface X { long m(); }

interface Y { int m(); }

interface Z extends X, Y {}

// Compiler error: no method is return type substitutable

interface Foo<T> { void m(T arg); }

interface Bar<T> { void m(T arg); }

interface FooBar<X, Y> extends Foo<X>, Bar<Y> {}

// Compiler error: different signatures, same erasure

**Lambda Expression**

Lambda Expression are the way through which we can visualize **functional programming** in the java object oriented world. Objects are the base of java programming language and we can never have a function without an Object, that’s why Java language provide support for using lambda expressions only with functional interfaces.

Since there is only one abstract function in the functional interfaces, there is no confusion in applying the lambda expression to the method. Lambda Expressions syntax is **(argument) -> (body)**. Now let’s see how we can write above anonymous Runnable using lambda expression.

Runnable r1 = () -> System.out.println("My Runnable");

Let’s try to understand what is happening in the lambda expression above.

* Runnable is a functionl interfce, that’s y v can use lambda expresion to create it’s instance.
* Since run() method takes no argument, our lambda expression also have no argument.
* Just like if-else blocks, we can avoid curly braces ({}) since we have a single statement in the method body. For multiple statements, we would have to use curly braces like any other methods.

**Why do we need Lambda Expression**

1. **Reduced Lines of Code**  
   One of the clear benefit of using lambda expression is that the amount of code is reduced, we have already seen that how easily we can create instance of a functional interface using lambda expression rather than using anonymous class.
2. **Sequential and Parallel Execution Support**

Another benefit of using lambda expression is that we can benefit from the Stream API sequential and parallel operations support.

To explain this, let’s take a simple example where we need to write a method to test if a number passed is prime number or not.Traditionally we would write it’s code like below. The code is not fully optimized but good for example purpose, so bear with me on this.

1. //Traditional approach
2. private static boolean isPrime(int number) {
3. if(number < 2) return false;
4. for(int i=2; i<number; i++){
5. if(number % i == 0) return false;
6. }
7. return true;

}

The problem with above code is that it's sequential in nature, if the number is very huge then it will take significant amount of time. Another problem with code is that there are so many exit points and it's not readable. Let's see how we can write the same method using lambda expressions and stream API.

//Declarative approach

private static boolean isPrime(int number) {

return number > 1

&& IntStream.range(2, number).noneMatch(

index -> number % index == 0);

}

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

For more readability, we can also write the method like below.

private static boolean isPrime(int number) {

IntPredicate isDivisible = index -> number % index == 0;

return number > 1

&& IntStream.range(2, number).noneMatch(

isDivisible);

}

If you are not familiar with IntStream, it's range() method returns a sequential ordered IntStream from startInclusive (inclusive) to endExclusive (exclusive) by an incremental step of 1.

noneMatch() method returns whether no elements of this stream match the provided predicate. It may not evaluate the predicate on all elements if not necessary for determining the result.

1. **Passing Behaviors into methods**

Let's see how we can use lambda expressions to pass behavior of a method with a simple example. Let's say we have to write a method to sum the numbers in a list if they match a given criteria. We can use Predicate and write a method like below.

public static int sumWithCondition(List<Integer> numbers, Predicate<Integer> predicate) {

return numbers.parallelStream()

.filter(predicate)

.mapToInt(i -> i)

.sum();

}

Sample usage:

//sum of all numbers

sumWithCondition(numbers, n -> true)

//sum of all even numbers

sumWithCondition(numbers, i -> i%2==0)

//sum of all numbers greater than 5

sumWithCondition(numbers, i -> i>5)

1. **Higher Efficiency with Laziness**

One more advantage of using lambda expression is the lazy evaluation, for example let's say we need to write a method to find out the maximum odd number in the range 3 to 11 and return square of it.

Usually we will write code for this method like this:

private static int findSquareOfMaxOdd(List<Integer> numbers) {

int max = 0;

for (int i : numbers) {

if (i % 2 != 0 && i > 3 && i < 11 && i > max) {

max = i;

} }

return max \* max;

}

Above program will always run in sequential order but we can use Stream API to achieve this and get benefit of Laziness-seeking. Let's see how we can rewrite this code in functional programming way using Stream API and lambda expressions.

public static int findSquareOfMaxOdd(List<Integer> numbers) {

return numbers.stream()

.filter(NumberTest::isOdd) //Predicate is functional interface and

.filter(NumberTest::isGreaterThan3) // we are using lambdas to initialize it

.filter(NumberTest::isLessThan11) // rather than anonymous inner classes

.max(Comparator.naturalOrder())

.map(i -> i \* i)

.get();

}public static boolean isOdd(int i) {

return i % 2 != 0;

}

public static boolean isGreaterThan3(int i){

return i > 3;

}

public static boolean isLessThan11(int i){

return i < 11;

}

If you are surprised with the double colon (::) operator, it's introduced in Java 8 and used for **method references**. Java Compiler takes care of mapping the arguments to the called method. It's short form of lambda expressions i -> isGreaterThan3(i) or i -> NumberTest.isGreaterThan3(i).

**Lambda Expression Examples**

Below I am providing some code snippets for lambda expressions with small comments explaining them.

() -> {} // No parameters; void result

() -> 42 // No parameters, expression body

() -> null // No parameters, expression body

() -> { return 42; } // No parameters, block body with return

() -> { System.gc(); } // No parameters, void block body

// Complex block body with multiple returns

() -> {

if (true) return 10;

else {

int result = 15;

for (int i = 1; i < 10; i++)

result \*= i;

return result;

}

}

(int x) -> x+1 // Single declared-type argument

(int x) -> { return x+1; } // same as above

(x) -> x+1 // Single inferred-type argument, same as below

x -> x+1 // Parenthesis optional for single inferred-type case

(String s) -> s.length() // Single declared-type argument

(Thread t) -> { t.start(); } // Single declared-type argument

s -> s.length() // Single inferred-type argument

t -> { t.start(); } // Single inferred-type argument

(int x, int y) -> x+y // Multiple declared-type parameters

(x,y) -> x+y // Multiple inferred-type parameters

(x, final y) -> x+y // Illegal: can't modify inferred-type parameters

(x, int y) -> x+y // Illegal: can't mix inferred and declared types

**Method and Constructor References**

A method reference is used to refer to a method without invoking it; a constructor reference is similarly used to refer to a constructor without creating a new instance of the named class or array type.

Examples of method and constructor references:

System::getProperty

System.out::println

"abc"::length

ArrayList::new

int[]::new