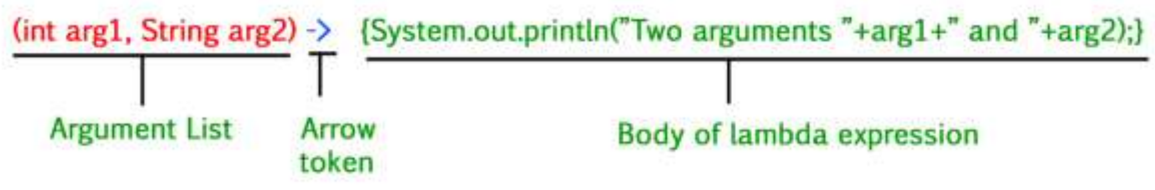
1. **Lambda Expressions:** Lambda expressions introduce a concise way to express instances of single-method interfaces (functional interfaces). They allow you to treat functionality as a method argument, or to create small, anonymous classes. Lambda expressions improve the readability and maintainability of code.

In **Java**, **Lambda** expressions basically express instances of functional interfaces (An interface with a single abstract method is called a functional interface). Lambda Expressions in Java are the same as lambda functions which are the short block of code that accepts input as parameters and returns a resultant value. Lambda Expressions are recently included in Java SE 8.

Functionalities of Lambda Expression in Java

Lambda Expressions implement the only abstract function and therefore implement functional interfaces lambda expressions are added in Java 8 and provide the below functionalities.

- Enable to treat functionality as a method argument, or code as data.
- A function that can be created without belonging to any class.
- A lambda expression can be passed around as if it was an object and executed on demand.


(int arg1, String arg2)
->
{System.out.println("Two arguments "+arg1+" and "+arg2);}
Argument List
Arrow token
Body of lambda expression

Lambda Expression Syntax

lambda operator -> body

Lambda Expression Parameters

There are three Lambda Expression Parameters are mentioned below:

1. Zero Parameter
2. Single Parameter
3. Multiple Parameters

1. Lambda Expression with Zero parameter

() -> System.out.println("Zero parameter lambda");

2. Lambda Expression with Single parameter

(p) -> System.out.println("One parameter: " + p);

It is not mandatory to use parentheses if the type of that variable can be inferred from the context

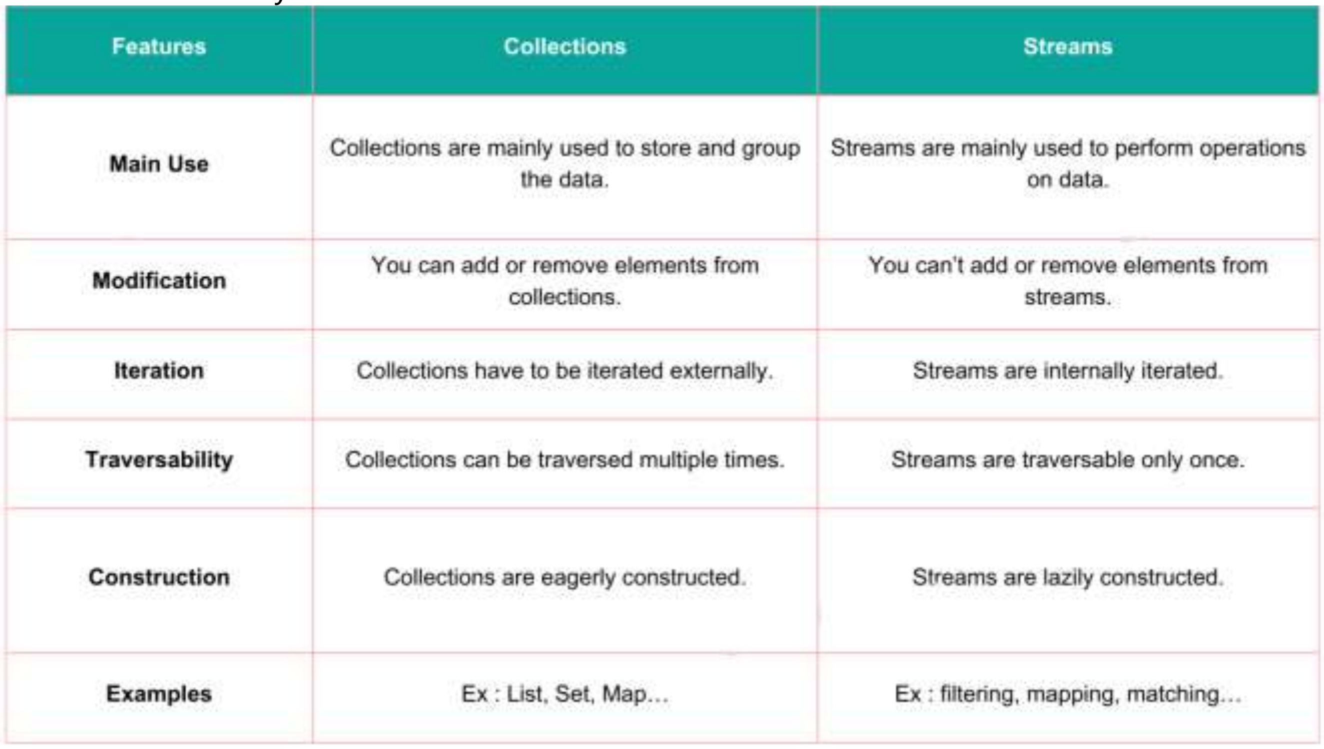
3. Lambda Expression with Multiple parameters

(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);

2. **Collection Streams:** Streams provide a new abstraction for processing sequences of elements. They allow for functional-style operations on collections, such as **filtering, mapping, reducing, and iterating**. Streams offer a declarative way to express complex data processing queries.

Collections vs Streams in Java

Here is the summary of the difference between Collections and Streams in Java:



Features	Collections	Streams
Main Use	Collections are mainly used to store and group the data.	Streams are mainly used to perform operations on data.
Modification	You can add or remove elements from collections.	You can't add or remove elements from streams.
Iteration	Collections have to be iterated externally.	Streams are internally iterated.
Traversability	Collections can be traversed multiple times.	Streams are traversable only once.
Construction	Collections are eagerly constructed.	Streams are lazily constructed.
Examples	Ex : List, Set, Map...	Ex : filtering, mapping, matching...

Main Use

Collections are used to store and group the data in a particular data structure like List, Set, or Map. Whereas Streams are used to perform complex data processing operations like filtering, matching, mapping, etc on stored data such as arrays, collections, or I/O resources.

That means, collections are mainly about data and streams are mainly about operations on data.

Example 1: List collection is used to store data

```
  // Creating an ArrayList of String using
     List<String> fruits = new ArrayList<>();
     // Adding new elements to the ArrayList
     fruits.add("Banana");
     fruits.add("Apple");
     fruits.add("mango");
     fruits.add("orange");
```

Example 2: Streams are used to perform operations like filtering, mapping, collection result, etc:

```
List<String> lines = Arrays.asList("java", "c", "python");

        List<String> result = lines.stream()       // convert list to stream
                .filter(line -> !"c".equals(line)) // we dont like c
                .collect(Collectors.toList());     // collect the output and
convert streams to a List

        result.forEach(System.out::println);
```

Modification

You can add or remove elements from collections. But, you can't add to or remove elements from streams. Stream consumes a source, performs operations on it, and returns a result. They don't modify even the source also.

For example:

```
        // Creating an ArrayList of String using
        List < String > fruits = new ArrayList < > ();
        // Adding new elements to the ArrayList
        fruits.add("Banana");
        fruits.add("Apple");
        fruits.add("Mango");
        fruits.add("Orange");
        fruits.add("Pineapple");
        fruits.add("Grapes");

        System.out.println(fruits);

        // Remove the element at index `5`
        fruits.remove(5);
        System.out.println("After remove(5): " + fruits);
```

In Streams, there are no such methods to add or remove elements.

Iteration

Streams perform iteration internally behind the scene for us (using the forEach() method). We just have to mention the operations to be performed on a source. On the other hand, we have to do the iteration externally over collections using loops.

Example 1: External iterations of Collections using for loops

```
// Creating an ArrayList of String using
List < String > fruits = new ArrayList < > ();
// Adding new elements to the ArrayList
fruits.add("Banana");
fruits.add("Apple");
fruits.add("mango");
fruits.add("orange");
fruits.add("Watermelon");
fruits.add("Strawberry");

System.out.println("\n=== Iterate using for loop with index ===");
for (int i = 0; i < fruits.size(); i++) {
    System.out.println(fruits.get(i));
}
```

```
System.out.println("=== Iterate using Java 8 forEach and lambda ===");
fruits.forEach(fruit - > {
    System.out.println(fruit);
});
```

Example 2: Internal iteration of Streams. No more for loops:

```
List<String> lines = Arrays.asList("java", "c", "python");

        List<String> result = lines.stream()       // convert list to stream
                .filter(line -> !"c".equals(line)) // we dont like c
                .collect(Collectors.toList());     // collect the output and
convert streams to a List

        result.forEach(System.out::println);
```

Traversability

Streams are traversable only once. If you traverse the stream once, it is said to be consumed. To traverse it again, you have to get a new stream from the source again. But, collections can be traversed multiple times.

```
List<Integer> numbers = Arrays.asList(4, 2, 8, 9, 5, 6, 7);
```

```
Stream<Integer> numbersGreaterThan5 = numbers.stream().filter(i -> i > 5);

//Traversing numbersGreaterThan5 stream first time

numbersGreaterThan5.forEach(System.out::println);

//Second time traversal will throw error

//Error : stream has already been operated upon or closed

numbersGreaterThan5.forEach(System.out::println);
```

Construction

Collections are eagerly constructed i.e. all the elements are computed at the beginning itself. But, streams are lazily constructed i.e. intermediate operations are not evaluated until the terminal operation is invoked.

Stream map()

Stream map(Function mapper) returns a stream consisting of the results of **applying the given function** to the elements of this stream. Stream map(Function mapper) is an intermediate operation. These operations are always lazy. Intermediate operations are invoked on a Stream instance and after they finish their processing, they give a Stream instance as output. Syntax :

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

```
where, R is the element type of the new stream.
Stream is an interface and T is the type
of stream elements. mapper is a stateless function
which is applied to each element and the function
returns the new stream.
```

3. **Interface Default Methods:** Java 8 introduced the ability to have default methods in interfaces. Default methods allow you to add new methods to interfaces without breaking existing implementations.

This feature facilitates the evolution of interfaces, particularly in evolving APIs.

Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

4. JDK 8 Concurrency Improvements: Java 8 includes enhancements to the concurrency API, such as the CompletableFuture class for handling asynchronous computations, and improvements to the parallel processing capabilities provided by streams.

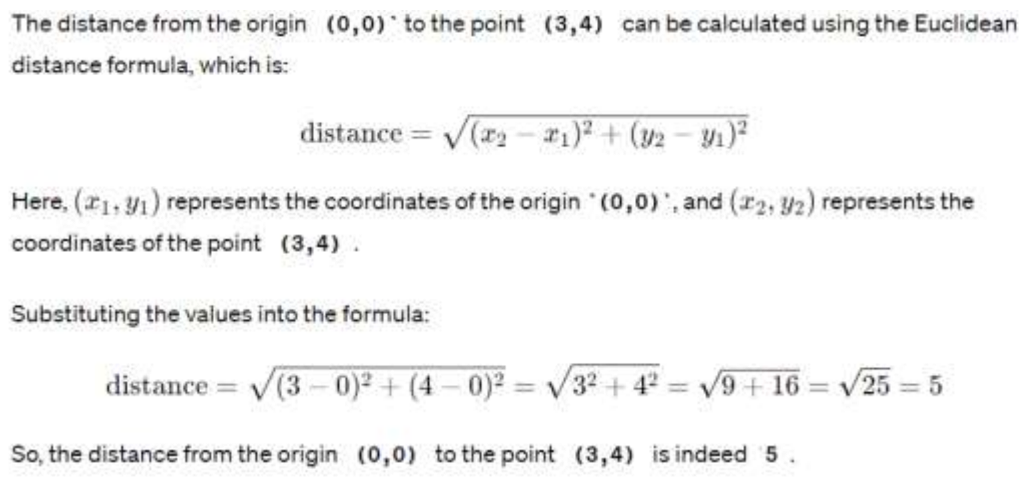
Java 8 introduced several enhancements to the concurrency utilities in the **java.util.concurrent** package. Some of the notable improvements include:

A. CompletableFuture: As demonstrated in the previous example, CompletableFuture is a powerful tool for asynchronous programming. It provides a way to perform computations asynchronously, with support for combining multiple asynchronous operations and handling their results.

B. Enhancements to Executors: Java 8 introduced new methods in the Executors class to create thread pools that better match the requirements of different scenarios. For example, **newWorkStealingPool()** creates a thread pool that uses a work-stealing algorithm, which can be more efficient for certain types of parallel tasks.

C. Parallel Streams: Java 8 introduced parallel streams, which allow you to execute stream operations concurrently on multiple threads. This enables easier parallelization of data processing tasks, such as filtering, mapping, and reducing elements in a collection.

D. StampedLock: StampedLock is a new type of lock introduced in Java 8. It provides a way to perform optimistic read locking and write locking with better performance than ReentrantReadWriteLock in certain scenarios.



The distance from the origin `(0,0)` to the point `(3,4)` can be calculated using the Euclidean distance formula, which is:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Here, (x_1, y_1) represents the coordinates of the origin `(0,0)`, and (x_2, y_2) represents the coordinates of the point `(3,4)`.

Substituting the values into the formula:

$$\text{distance} = \sqrt{(3-0)^2 + (4-0)^2} = \sqrt{3^2 + 4^2} = \sqrt{9 + 16} = \sqrt{25} = 5$$

So, the distance from the origin `(0,0)` to the point `(3,4)` is indeed `5`.

E. Atomic Updates: Java 8 added new methods to the AtomicInteger, AtomicLong, and AtomicReference classes for performing atomic updates on variables. For example, methods like updateAndGet() and accumulateAndGet() allow you to atomically update the value of an AtomicInteger or AtomicLong using a lambda expression or an accumulator function.

F. Concurrent Accumulators: Java 8 introduced the LongAccumulator and DoubleAccumulator classes, which provide support for concurrent accumulation of values using custom functions. These classes can be useful for parallel processing tasks where multiple threads need to update a shared result.

G. ForkJoinPool Improvements: Java 8 made improvements to the ForkJoinPool class, which is used by the parallel streams framework and other parallel processing utilities. These improvements include better handling of common pool initialization, dynamic adjustment of parallelism levels, and improved work-stealing algorithms.

These concurrency improvements in Java 8 make it easier to write efficient and scalable concurrent code, enabling developers to take advantage of multi-core processors and parallel execution models more effectively.

5. JDK 8 Java IO Improvements: Java 8 introduced various improvements to the Java IO (Input/Output) API, such as the Files class for working with files and directories, the Path interface for representing file system paths, and enhancements to existing IO classes.

6. JSR 223 Feature: JSR 223, or "Scripting for the Java Platform", defines an API for integrating scripting languages into Java applications. It allows Java applications to interact with scripts written in languages like JavaScript, Ruby, Python, etc., enabling better integration of scripting capabilities into Java applications.

7. Other Core Java/JDK 8 Features: Some additional features and improvements in Java 8 and beyond include:

- **Date and Time API:** A new API for handling dates and times, which addresses the shortcomings of the old **java.util.Date** and **java.util.Calendar** classes.
- **Optional class:** A container object that may or may not contain a non-null value. It helps to reduce **NullPointerExceptions** in code.
- **Type Annotations:** Java 8 introduced type annotations, which allow you to apply annotations to any type use, not just declarations.

- **Nashorn JavaScript Engine:** Java 8 includes a new JavaScript engine called **Nashorn**, which provides improved performance and compliance with **ECMAScript** standards.