the Stream API brings totally new ways for working with collections. Once you are familiar with it, you will love using it, as it makes you write code more natural, more succinct, more readable and most importantly, your productivity will go up incredibly. Trust me!

Let's start by looking at some code examples to understand how the Stream API radically changes the way we work with collections. Don't worry if you don't fully understand the code right away.

Keep reading…

Suppose that we have the `Student` class as shown below:

```
class Student implements Comparable< Student > {

	private String name;

	private int score;

	public Student(String name, int score) {

		this.name = name;

		this.score = score;

	}

	// getters and setters....

	public String toString() {

		return this.name + " - " + this.score;

	}

	public int compareTo(Student another) {

		return another.getScore() - this.score;

	}

}
```

and a list of students:

```
List< Student > listStudents = new ArrayList<>();

listStudents.add(new Student("Alice", 82));

listStudents.add(new Student("Bob", 90));

listStudents.add(new Student("Carol", 67));
```

```
listStudents.add(new Student("David", 80));
listStudents.add(new Student("Eric", 55));
listStudents.add(new Student("Frank", 49));
listStudents.add(new Student("Gary", 88));
listStudents.add(new Student("Henry", 98));
listStudents.add(new Student("Ivan", 66));
listStudents.add(new Student("John", 52));
```

We are required to do some calculations on this list.

First, find the students whose scores are greater than or equal to 70.

A non-stream solution would look like this:

```
// find students whose score >= 70
List< Student > listBadStudents = new ArrayList<>();
for (Student student : listStudents) {
      if (student.getScore() >= 70) {
            listBadStudents.add(student);
      }
}
for (Student student : listBadStudents) {
      System.out.println(student);
}
```

With the Stream API, we can replace the above code with the following:

```
// find students whose score >= 70
List< Student > listGoodStudents = listStudents.stream()
                                 .filter(s -> s.getScore() >= 70)
                                 .collect(Collectors.toList());
listGoodStudents.stream().forEach(System.out::println);
```

Don’t worry if you don’t understand the code. Just see the differences between non-stream code and stream-based code.

Second, calculate average score of all students. A trivial solution would look like this:

```
// calculate average score of all students

double sum = 0.0;

for (Student student : listStudents) {

      sum += student.getScore();

}

double average = sum / listStudents.size();

System.out.println("Average score: " + average);
```

And here's a stream-based version:

```
// calculate average score of all students

double average = listStudents.stream()

                  .mapToInt(s -> s.getScore())

                  .average().getAsDouble();

System.out.println("Average score: " + average);
```

That's it!

So what the differences do you see between the non-stream code and the stream-based code?

They look totally different, right? Do you notice that the stream-based version looks more natural, something like a query, right? But that's not all.

Continue reading and you'll see how streams are really powerful and flexible.

* What is a Stream?

A stream represents a sequence of elements supporting sequential and parallel aggregate operations. Since Java 8, we can generate a stream from a collection, an array or an I/O channel.

Every collection class now has the **`stream()`** method that returns a stream of elements in the collections:

```
Stream stream = listStudents.stream();
```

Obtaining a stream from an array:

```
int[] arrayIntegers = {1, 8, 2, 3, 98, 11, 35, 91};
```

```
IntStream streamIntegers = Arrays.stream(arrayIntegers);
```

Obtaining a stream from a file:

```
BufferedReader bufferReader = new BufferedReader(new
FileReader("students.txt"));

Stream streamLines = bufferReader.lines();
```

Operations can be performed on a stream are categorized into intermediate operations and terminal operations. We'll see details of these operations shortly. Consider the following code:

```
List< Student > top3Students = listStudents.stream()

                        .filter(s -> s.getScore() >= 70)

                        .sorted()

                        .limit(3)

                        .collect(Collectors.toList());

System.out.println("Top 3 Students by Score:");

top3Students.forEach(s -> System.out.println(s));
```

This code can be read as: select top 3 students whose scores >= 70, and sort them by score in descending order (the natural ordering of the `Student` class). Here we can see the following intermediate operations: `filter`, `sorted` and `limit`; and the terminal operation is `collect`.

As you can see, the operations on a stream can be chained together (intermediate operations) and end with a terminal operation. Such a chain of stream operations is called stream pipeline.

* Stream Pipeline:

We can say that a stream is a pipeline of aggregate operations that can be evaluated. A pipeline consists of the following parts:

- a source: can be a collection, an array or an I/O channel.

- zero or more intermediate operations which produce new streams, such as `filter`, `map`, `sorted`, etc.

- a terminal operation that produces a non-stream result such as a primitive value, a collection, or void (such as the `forEach` operation).

* Intermediate Operations:

An intermediate operation processes over a stream and return a new stream as a response. Then we can execute another intermediate operation on the new stream, and so on, and finally execute the terminal operation.

One interesting point about intermediate operations is that they are lazily executed. That means they are not run until a terminal operation is executed.

The Stream API provides the following common intermediate operations:

- `map()`
- `filter()`
- `sorted()`
- `limit()`
- `distinct()`

For a full list of intermediate operations, consult the Stream Javadoc.

* Terminal Operations:

A stream pipeline always ends with a terminal operation, which returns a concrete type or produces a side effect. For instances, the `collect` operation produces a collection; the `forEach` operation does not return a concrete type, but allows us to add side effect such as print out each element.

Unlike lazily-executed terminate operations, a terminal operation is always eagerly executed. The common terminal operations provided by the Stream API include:

- `collect()`
- `reduce()`
- `forEach()`

See the Stream Javadoc for a complete list of terminal operations supported.

* Parallel Streams:

The powerful feature of streams is that stream pipelines may execute either sequentially or in parallel. All collections support the `parallelStream()` method that returns a possibly parallel stream:

```
Stream parallelStream = listStudents.parallelStream();
```

When a stream executes in parallel, the Java runtime divides the stream into multiple sub streams. The aggregate operations iterate over and process these sub streams in parallel and then combine the results.

The advantage of parallel streams is performance increase on large amount of input elements, as the operations are executed currently by multiple threads on a multi-core CPU.

For example, the following code may execute stream operations in parallel:

```
listStudents.parallelStream()
            .filter(s -> s.getScore() >= 70)
            .sorted()
            .limit(3)
            .forEach(System.out::println);
```

The Collection's `stream()` method returns a sequential stream. We can convert a sequential stream to a parallel stream by calling the **`parallel()`** method on the current stream. The following example shows a stream executes the `sorted` operation sequentially, and then execute the `filter` operation in parallel:

```
listStudents.stream()
            .sorted()
            .parallel()
            .filter(s -> s.getScore() >= 70)
            .forEach(System.out::println);
```

* Streams and Lambda Expressions:

As you can see in the above examples, Lambda expressions can be used as arguments in aggregate functions. This allows us to write code more flexibility and more compact. Remember that the parameter in the Lambda expression is implicitly the object being processed in the stream.

Consider the following example:

```
listStudents.stream()
      .sorted()
      .filter(s -> s.getScore() >= 70)
      .forEach(System.out::println);
```

Here, we use a Lambda expression in the `filter` operation and a static method reference in the `forEach` operation. The `s` parameter is of type `Student` because the stream is a sequence of `Student` objects.

NOTE: Some operations can transform a stream of type A to a stream of type B, such as the map operation in the following example:

```
listStudents.stream()
      .filter(s -> s.getScore() >= 70)
```

```
        .map(s -> s.getName())

        .sorted()

        .forEach(name -> System.out.println(name));
```

In Lambda expressions used with the `filter` and `map` operations, the `s` parameter is of type `Student`. However the map operation produces a stream of Strings, so the `name` parameter in the Lambda expression in the `forEach` operation is of type `String`. So pay attention to this kind of transformation when using Lambda expressions.

* Streams vs. Collections:

A collection is a data structure that holds elements. Each element is computed before it actually becomes a part of that collection.

On the other hand, a stream is not a data structure. A stream is a pipeline of operations that compute the elements on-demand. Though we can create a stream from a collection and apply a number of operations, the original collection doesn't change. Hence streams cannot mutate data.

And a key characteristic of streams is that they can transform data, as operations on a stream can produce another data structure, like the `map` and `collect` operation as shown in the above examples.

Now, let's look closer at these common aggregate functions in details.

Before begin, let's see the data structure used in the examples. Given the following `Person` class:

```
public class Person implements Comparable< Person > {

        private String firstName;

        private String lastName;

        private String email;

        private Gender gender;

        private int age;

        public Person() {

        }

        public Person(String firstName, String lastName,

                        String email, Gender gender, int age) {
```

```
            this.firstName = firstName;

            this.lastName = lastName;

            this.email = email;

            this.gender = gender;

            this.age = age;

      }

      // geters and setters go here...

      public int compareTo(Person another) {

            return this.age - another.getAge();

      }

      public String toString() {

            return this.firstName + " " + this.lastName;

      }

}
```

Note that the getters and setters are removed for brevity (you should implement them completely), and the natural ordering of this class is based on age of the person (see its `compareTo()` method). Also, the `toString()` method returns the name of the person in form of first name followed by last name, so printing a `Person` object will show its full name.

The `gender` property is an enum which is declared as follows:

```
public enum Gender { MALE, FEMALE }
```

The sample data is hardcoded as follows:

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

listPersons.add(new Person("Alice", "Brown", "alice@gmail.com",
Gender.FEMALE, 26));

listPersons.add(new Person("Bob", "Young", "bob@gmail.com",
Gender.MALE, 32));

listPersons.add(new Person("Carol", "Hill", "carol@gmail.com",
Gender.FEMALE, 23));

listPersons.add(new Person("David", "Green", "david@gmail.com",
Gender.MALE, 39));
```

```
listPersons.add(new Person("Eric", "Young", "eric@gmail.com",
Gender.MALE, 26));

listPersons.add(new Person("Frank", "Thompson", "frank@gmail.com",
Gender.MALE, 33));

listPersons.add(new Person("Gibb", "Brown", "gibb@gmail.com",
Gender.MALE, 27));

listPersons.add(new Person("Henry", "Baker", "henry@gmail.com",
Gender.MALE, 30));

listPersons.add(new Person("Isabell", "Hill", "isabell@gmail.com",
Gender.FEMALE, 22));

listPersons.add(new Person("Jane", "Smith", "jane@gmail.com",
Gender.FEMALE, 24));
```

Okay, let's examine the aggregate functions that are intermediate operations first. For terminal operations, we simple use the `forEach` operation that print out the current element in the result stream.

* the filter operation

The **`filter()`** operation returns a new stream that consists elements matching a given condition which is typically a boolean test in form of a Lambda expression.

The following example lists only male persons:

```
listPersons.stream()

      .filter(p -> p.getGender().equals(Gender.MALE))

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

Output:

```
Bob Young

David Green

Eric Young

Frank Thompson

Gibb Brown

Henry Baker
```

The following code shows only female who are under 25:

```
listPersons.stream()

      .filter(p -> p.getGender().equals(Gender.FEMALE) && p.getAge()
<= 25)

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

Result:

```
Carol Hill

Isabell Hill

Jane Smith
```

* the map operation

The map operation returns a new stream consisting of elements which are the results of applying a given function to the elements of the current stream. For example, converting a stream of Objects to a stream of String or a stream of primitive numbers.

The Stream API provides 4 methods for the map operation:

- **`map()`**: transforms a stream of objects of type `T` to a stream of objects of type `R`.
- **`mapToInt()`**: transforms a stream of objects to a stream of `int` primitives.
- **`mapToLong()`**: transforms a stream of objects to a stream of `long` primitives.
- **`mapToDouble()`**: transforms a stream of objects to a stream of `double` primitives.

The following code maps each person to his/her respective email address:

```
listPersons.stream()

      .map(p -> p.getEmail())

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

Output:

```
alice@gmail.com

bob@gmail.com

carol@gmail.com

david@gmail.com

eric@gmail.com
```

frank@gmail.com

gibb@gmail.com

henry@gmail.com

isabell@gmail.com

jane@gmail.com

The following example maps each person to his/her age:

```
listPersons.stream()
        .mapToInt(p -> p.getAge())
        .forEach(age -> System.out.print(age + " - "));
```

Output:

```
26 - 32 - 23 - 39 - 26 - 33 - 27 - 30 - 22 - 24 -
```

The following example maps each person to his/her first name in uppercase:

```
listPersons.stream()
        .map(p -> p.getFirstName().toUpperCase())
        .forEach(System.out::println);
```

Output:

```
ALICE
BOB
CAROL
DAVID
ERIC
FRANK
GIBB
HENRY
ISABELL
JANE
```

* the sorted operation:

The Stream API provides two overloads of the sorted operation that returns a new stream consisting elements sorted according to natural order or by a specified comparator:

- **`sorted()`**: sorts by natural order
- **`sorted(comparator)`**: sorts by a comparator

The following example returns a stream of persons who are sorted by their age into ascending order:

```
listPersons.stream()

      .sorted()

      .forEach(p -> System.out.println(p + " - " + p.getAge()));
```

Look at the `compareTo()` method in the `Person` class, we see that the natural ordering is based on age:

```
public int compareTo(Person another) {

      return this.age - another.getAge();

}
```

Output of the above code:

```
Isabell Hill - 22

Carol Hill - 23

Jane Smith - 24

Alice Brown - 26

Eric Young - 26

Gibb Brown - 27

Henry Baker - 30

Bob Young - 32

Frank Thompson - 33

David Green - 39
```

The following code shows how to use a specified comparator to return a stream of persons who are sorted by their last name:

```
listPersons.stream()

      .sorted((p1, p2) ->
p1.getLastName().compareTo(p2.getLastName()))

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

Output:

```
Henry Baker

Alice Brown

Gibb Brown

David Green

Carol Hill

Isabell Hill

Jane Smith

Frank Thompson

Bob Young

Eric Young
```

* the distinct operation:

The **`distinct()`** operation returns a stream consisting of the distinct elements (no duplicates) by comparing objects via their `equals()` method.

The following example returns a stream of distinct numbers from an array source:

```
int[] numbers = {23, 58, 12, 23, 17, 29, 99, 98, 29, 12};

Arrays.stream(numbers).distinct().forEach(i -> System.out.print(i + "
"));
```

Output:

```
23 58 12 17 29 99 98
```

Combining with the map and sorted operations, the following example shows distinct last names of the persons in the above list, and sorts them by alphabetic order:

```
listPersons.stream()

      .map(p -> p.getLastName())
```

```
        .distinct()

        .sorted()

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

Output:

```
Baker

Brown

Green

Hill

Smith

Thompson

Young
```

* the limit operation:

The **`limit()`** operation returns a stream containing only a specified number of elements. Combining with the `sorted()` operation, the following example shows top 5 youngest persons:

```
listPersons.stream()

        .sorted()

        .limit(5)

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

Output:

```
Isabell Hill

Carol Hill

Jane Smith

Alice Brown

Eric Young
```

* the skip operation:

The **`skip()`** operation returns a stream containing the remaining elements after discarding the first n elements of the stream.

Combining with the sorted and map operations, the following example finds the oldest age of the persons above:

```
System.out.print("The oldest age: ");

listPersons.stream()

      .sorted()

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

      .skip(listPersons.size() - 1)

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

Output:

```
The oldest age: 39
```

As you can see, we can combine (chain) some aggregate functions together to achieve desired results. Such chaining is very common with streams operations. That makes streams powerful and flexible. I bet you will definitely love using it.

Now, let's discover the terminal operations provided by the Stream API in details. Remember the following two key characteristics of terminal operations:

- They can return a primitive value (`boolean`or `long`), a concrete type (`Optional` value object), or void (creating side effect).
- They are eagerly executed, and a terminal operation is always the last operation in a Stream pipeline.

Note that the following examples are still based on the sample data (a list of `Person` objects) in the previous email.

* The allMatch operation:

The **`allMatch()`** operation answers the question: *Do all elements in the stream meet this condition?* It returns true if and only if all elements match a provided predicate, otherwise return false.

This is a short-circuiting terminal operation because the operation stops immediately if any unmatched element is found (just like short-circuit behavior of the AND operator).

The following example checks if all persons are male:

```
boolean areAllMale = listPersons.stream()

                          .allMatch(p ->
p.getGender().equals(Gender.MALE));

System.out.println("Are all persons male? " + areAllMale);
```

Result:

```
Are all persons male? false
```

The following example check if all people using GMail:

```
boolean useGMail = listPersons.stream()

                          .allMatch(p ->
p.getEmail().endsWith("gmail.com"));

System.out.println("Does everyone use GMail? " + useGMail);
```

Result:

```
Does everyone use GMail? true
```

Note that if the stream is empty, the method returns true and the predicate is not evaluated.

* The anyMatch operation:

The **`anyMatch()`** operation returns true if any element in the stream matches a provided predicate. In other words, it answers the following question: *Is there any element that meets this condition?*

The following example checks whether the list has any female:

```
boolean anyFemale = listPersons.stream()

                  .anyMatch(p ->
p.getGender().equals(Gender.FEMALE));

System.out.println("Is there any female? " + anyFemale);
```

Result:

```
Is there any female? true
```

The following example checks whether the list has any teenager (age under 20):

```
boolean anyTeenager = listPersons.stream()

                  .anyMatch(p -> p.getAge() < 20);
```

```
System.out.println("Is there any teenager? " + anyTeenager);
```

Output:

```
Is there any teenager? false
```

* The noneMatch operation:

In contrast to the `allMatch()` operation, the **`noneMatch()`** operation returns true if no elements in the stream match a provided predicate. In other words, it answers the question: *Does no element meet this condition?*

The following example checks if no none uses Yahoo email:

```
boolean noYahooMail = listPersons.stream()

                   .noneMatch(p ->
p.getEmail().endsWith("yahoo.com"));

System.out.println("No one uses Yahoo mail? " + noYahooMail);
```

Result:

```
No one uses Yahoo mail? true
```

The following example answers the question: Does anyone come from the Hill family?

```
boolean noHill = listPersons.stream()

                   .noneMatch(p -> p.getLastName().equals("Hill"));

System.out.println("No one comes from Hill family? " + noHill);
```

Output:

```
   No one comes from Hill family? false
```

* The collect operation:

The **`collect()`** operation accumulates elements in a stream into a container such as a collection. It performs mutable reduction operation in which the reduced (final) value is a mutable result container such as an `ArrayList`. This method takes a `Collector` implementation that provides useful reduction operations.
The `Collectors` class is a common implementation in the JDK. And we are going to see how it is used in the following examples.

The following example accumulates emails of the persons into a list collection:

```
List< String > listEmails = listPersons.stream()

                        .map(p -> p.getEmail())

                        .collect(Collectors.toList());

System.out.println("List of Emails: " + listEmails);
```

Output:

```
List of Emails:
[alice@gmail.com, bob@gmail.com, carol@gmail.com, david@gmail.com, er
ic@gmail.com, frank@gmail.com, gibb@gmail.com, henry@gmail.com, isabe
ll@gmail.com, jane@gmail.com]
```

We can specify exactly which type of collection as the result. For example, the following code collects emails into a `TreeSet`:

```
Set< String > setEmails = listPersons.stream()

                        .map(p -> p.getEmail())

                        .collect(Collectors.toCollection(TreeSet::new
));
```

Output:

```
Set of Emails:
[alice@gmail.com, bob@gmail.com, carol@gmail.com, david@gmail.com, er
ic@gmail.com, frank@gmail.com, gibb@gmail.com, henry@gmail.com, isabe
ll@gmail.com, jane@gmail.com]
```

The following example groups the person by gender:

```
Map< Gender, List< Person > > byGender = listPersons.stream()

                    .collect(Collectors.groupingBy(p ->
p.getGender()));

System.out.println("Groups by gender:\n" + byGender);
```

Output:

```
{FEMALE=[Alice Brown, Carol Hill, Isabell Hill, Jane Smith],
MALE=[Bob Young, David Green, Eric Young, Frank Thompson, Gibb Brown,
Henry Baker]}
```

The following example accumulates first names and concatenates them into a String, separated by commas:

```
String firstNames = listPersons.stream()
```

```
                        .map(p -> p.getFirstName())

                        .collect(Collectors.joining(", "));
```

Result:

```
      First Names: Alice, Bob, Carol, David, Eric, Frank, Gibb, Henry,
Isabell, Jane
```

Consult the Collectors Javadoc for more useful mutable reduction operations.

* The count operation:

The **`count()`** operation simply returns total number of elements in the stream. The following example finds how many people are male:

```
      long totalMale = listPersons.stream()

                        .filter(p -> p.getGender().equals(Gender.MALE))

                        .count();

      System.out.println("Total male: " + totalMale);
```

Output:

```
         Total male: 6
```

* The forEach operation:

The **`forEach()`** operation performs an action for each element in the stream, thus creating a side effect, such as print out information of each female person as shown in the following example:

```
      System.out.println("People are female:");

      listPersons.stream()

            .filter(p -> p.getGender().equals(Gender.FEMALE))

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

Output:

```
      People are female:

      Alice Brown
```

```
Carol Hill

Isabell Hill

Jane Smith
```

* The min operation:

The **`min(comparator)`** is a special reduction operation that returns the minimum element in the stream according to the provided comparator. It returns an `Optional` which is a container object that contains the value.

For example, the following code snippet finds the youngest female person in the list:

```
Optional< Person > min = listPersons.stream()

                  .filter(p -> p.getGender().equals(Gender.FEMALE))

                  .min((p1, p2) -> p1.getAge() - p2.getAge());

if (min.isPresent()) {

      Person youngestGirl = min.get();

      System.out.println("The youngest girl is: "

                        + youngestGirl + " (" + youngestGirl.getAge()
+ ")");

}
```

Output:

```
The youngest girl is: Isabell Hill (22)
```

* The max operation:

Similar to the `min()` operation, the **`max()`** is a special reduction operation that returns the maximum element in the stream according to the specified comparator. The following example finds the oldest male person in the list:

```
Optional max = listPersons.stream()

                  .filter(p -> p.getGender().equals(Gender.MALE))

                  .max((p1, p2) -> p1.getAge() - p2.getAge());

if (max.isPresent()) {
```

```
        Person oldestMan = max.get();

        System.out.println("The oldest man is: "

                           + oldestMan + " (" + oldestMan.getAge() +
")");

}
```

Result:

```
The oldest man is: David Green (39)
```

* The reduce operation:

The Stream API provides three versions of **`reduce()`** methods which are general reduction operations. Let's look at each version.

- Version #1: **`Optional< T > reduce(BinaryOperator< T > accumulator)`**

This method performs a reduction on the elements of the stream, using an associative accumulation function, and returns an `Optional` object describing the reduced value. For example, the following code accumulates first names of all persons into a String:

```
Optional< String > reducedValue = listPersons.stream()

                                  .map(p -> p.getFirstName())

                                  .reduce((name1, name2) -> name1 + ", "
+ name2);

if (reducedValue.isPresent()) {

        String names = reducedValue.get();

        System.out.println(names);

}
```

Output:

```
Alice, Bob, Carol, David, Eric, Frank, Gibb, Henry, Isabell, Jane
```

- Version #2: **`T reduce(T identity, BinaryOperator< T > accumulator)`**

This method is similar to the version #1, but it returns the reduced value of the specified type `T`. The `identity` value must be an identity value for the accumulator function, which means it does not affect the result of accumulation. The following example calculates sum of numbers in a stream:

```
int[] numbers = {123, 456, 789, 246, 135, 802, 791};

int sum = Arrays.stream(numbers).reduce(0, (x, y) -> (x + y));

System.out.println("sum = " + sum);
```

Output:

```
sum = 3342
```

- Version #3:

```
U reduce(U identity,

         BiFunction< U,? super T, U > accumulator,

         BinaryOperator< U > combiner)
```

This is the most general reduction method that performs a reduction on elements of the stream, using the provided identity, accumulator and combiner.

The identity element is both an initial seed value for the reduction and a default result if there are no input elements.

The accumulator function takes a partial result and the next element, and produces a new partial result

The combiner function combines two partial results to produce a new partial result (it is necessary in parallel reductions).

The following example shows how this general reduction operation is used to accumulate numbers to calculate sum of them:

```
int[] numbers = {123, 456, 789, 246, 135, 802, 791};

int sum = Arrays.stream(numbers).reduce(0, (x, y) -> (x + y),
Integer::sum);

System.out.println("sum = " + sum);
```

That’s all about common terminal operations provided by the Stream API. For in-depth information about stream operations, see its Javadoc.