Java Fundamentals

Java is similar to C# in many ways, which means it's not too big of a lift to learn one language when you already know the other.

What's the deal with C# and Java?

Java was first released in 1996 (about four years before C#). It was created by researchers at Sun Microsystems (which was eventually acquired by Oracle). The language was based on the older C++ language's syntax, using curly brackets for code blocks and short, lowercase type names like int, and the familiar for-loop syntax, all of which C# also shares today. One main advancement over C++ was that in Java when you're finished using an object you created, Java will remove it from memory automatically. (This is called garbage collection.) C++, on the other hand, requires programmers to manually "delete" their objects. Manual deletion is incredibly difficult

to manage, and as such the vast majority of C++ programs often have bugs in them called memory leaks. Java solved this problem once and for all.

Initially Java ran primarily in the browser. Users had to download the Java extension, and then little Java programs called “applets” could be run in the browser. Because it was a browser-only language, the language did not include the ability to interact with the operating system. There was no way to read and write files, for example.

Soon after Java was introduced, Microsoft was looking for a new language that would become the primary language for building Windows applications that ran outside the browser. They liked the Java language, but needed their language to interact with the operating system and allow programs to read and write files. They initially released a version of Java for Windows that was quite different from the official Java standard Sun had created. Sun filed a lawsuit, stopping Microsoft from changing the language. And so Microsoft built their own language, starting with one very similar to Java, but with the operating system additions. They tweaked the language just a bit here and there. (You’ll see some differences in this lesson.) And then they added in full support for interacting with the operating system.

They needed a new name for their new language; the original “C” language was still quite popular, and its successor, C++, was easily the most popular language at the time. They decided to go with the C theme but add on a different symbol. They went with the hash symbol, also called a pound sign, or a number sign. But for pronunciation, instead they went with the musical notation name for the symbol, “Sharp”. And hence C# was born as sort of a descendent of Java.

Interestingly, Sun eventually saw how Java could be useful outside of the browser, and they finally added operating system features to Java, including file input and output. Then after that, interest in running Java applets in the browser waned as people were more interested in Macromedia Flash (later owned by Adobe) and after that HTML5 with JavaScript.

Since then Java and C# have more-or-less copied each other, adding new features that are similar. Today they can both be used as the back end programming, leaving the front end to JavaScript and TypeScript.

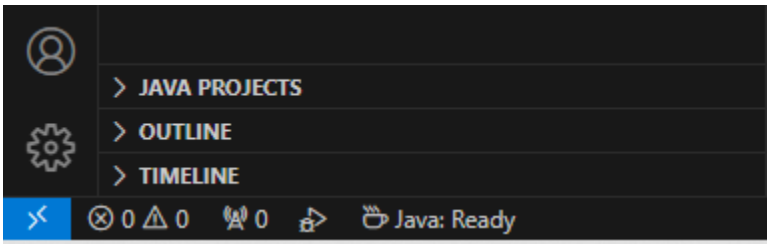
But they do have their differences. One major difference is that when Google introduced the Android phone, they chose Java as its language. Since then it’s also become popular for embedded programming and Internet-of-Things (IoT) programming. C#, on the other hand, works great for Windows programming and has been the language of choice for gaming, especially with the Unity gaming engine. Both

languages are also popular in business applications. Typically to run C# applications in a full-stack environment, you'll need a Windows back-end server. For Java full stack applications, you'll typically use a Linux back-end server, which can cost significantly less due to Windows licensing requirements.

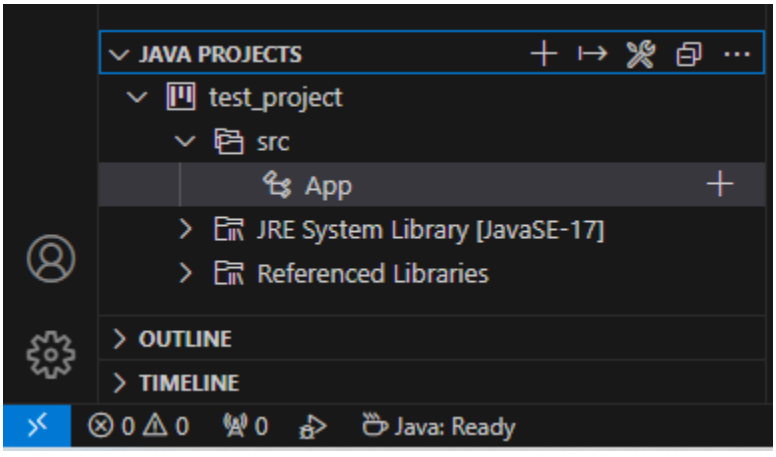
Starting a New Project

Follow the instructions for creating a new project here: [Java Setup Guide]

By default, VS Code displays the folder structure in your application in the "Explorer" window in the upper left. However, Java developers prefer to work in a mode called Project, which gives you a slightly different perspective of the files in your project. The Extension Pack for Java plugin provides a Project view for your Java applications. You can find it in the lower-left corner:



Click on Java Projects to open this view:



TIp: You can resize the window to be taller by dragging the top of it upward.

Notice that instead of displaying the full filename App.java under the src folder, you only see the name of the class App.

Important: Unlike C#, in Java, each class typically goes in its own file. The file name must match the name of the class, but with .java added. So for example, if you have a class called Rectangle, then the file must be called Rectangle.java.

The App.java starter code

Here's the code inside App.java; it should look similar to C#:

```
public class App {
    public static void main(String[] args) throws Exception {
        System.out.println("Hello, World!");
    }
}
```

The usual things are there, including matching curly brackets, and semicolons. In that regard, Java looks almost identical to C#.

This class is considered public, which means it can be accessed from code outside the file. (There's more to this, but that's the start of it.)

The class contains a static method called main. This is the "entry point" when you run the Java application. It's where your program begins.

This main method doesn't return anything, hence it gets a return type of void, just like with C#. Also like C#, the main method takes as parameters an array of strings called args.

Tip: Notice the S in String is uppercased, unlike in C#. (We explain the difference later in this document.)

Next is the odd looking "throws Exception". In Java, exception handling plays an important part of coding. We'll look at more on that later.

And finally, instead of using Console.WriteLine like in C#, we use System.out.println. System is a class inside Java's framework, and it contains a static member called out. That member in turn contains a function called println, which works very similarly to WriteLine in C#.

Variables, If Statements, and Loops

Java shares many common types with C# and other languages, including integers, floating point numbers, booleans, and strings.

Later we look at important differences between Java and C# regarding types, but first let's just create some variables in Java. Here are the basic types: 8 lowercase types called "primitive types" and String, which is always capitalized.

Integer types	**byte** - 1 byte of memory **short** - 2 bytes of memory **int** - (most common) 4 bytes **long** - 8 bytes
Decimal types	**float** - 4 bytes **double** - (most common) 8 bytes
Other primitives	**boolean** - true/false **char** - a single character
String type	**String**

Here are some brief examples.

The "if" statement works very similarly as in C#.

```
char dayOfWeek = 'M'; // Note: char literals use single quotes
boolean holiday = false;

if (dayOfWeek == 'S' || dayOfWeek == 'U' || holiday) {
    System.out.println("Take the day off!");
} else {
    System.out.println("Let's get to work!");
}
```

First we'll sum the numbers from 1 to 20 and print them out. As you can see, this code is very similar to C#.

```
int sum = 20;
System.out.println("Let's add the numbers from 1 to 20!");
```

```
for (int x = 1; x <= 20; x++) {
    sum += x;
}
System.out.println(sum);
```

You can see we used the += assignment operator to add x to sum and store the value back in sum, just like with C#. (The others are also available, -=, *=, and /= among others.)

Now let's do some math; we have all the same operations we have in C#.

```
int num = 10;
int next = num * 2 + 5;
System.out.println(next);
System.out.println(next * 3);
System.out.println(next / 6); // Look at this one closely. Does it
print any decimal parts?
```

Let's do the same but with the double type, which is a double-precision floating point (and what's use most often for floating point numbers.)

```
double number = 10.0;
double nextNumber = number * 2 + 5;
System.out.println(nextNumber);
System.out.println(nextNumber * 3);
System.out.println(nextNumber / 6); // Look at this one again
```

Next let's create a string, and see a loop. Here we'll loop through the individual characters in a string using a for loop that looks basically the same as in C#:

```
String name = "Doctor Dexian";
for (int x = 0; x < name.length(); x++) {
    System.out.println(name.charAt(x));
}
```

One important difference from C# is that in Java we don't access the individual characters in the string with bracket notation (e.g. name[3]) as we can in C#. Instead we call the charAt method.

In C#, we have a handy foreach keyword that lets us loop through strings, arrays, and collections. Java has something similar, but it doesn't work on strings. We'll look at it shortly when we cover arrays.

Java also has a while loop that works just like with C#. We'll demonstrate that later today when we show how to ask the user for input.

Important Differences Between Types

But there are some differences in what types are available to you.

C# uses a runtime, which is part of .NET, that already knows about the basic types, including ints, booleans, and strings. These types have names like Int32, Boolean, and String, and they all start with a capital letter. The C# compiler includes many type names built into the language that are simply alias names to these .NET types (and they all start with a lowercase letter). For example, the type "int" is just an alias for Int32. All of these are classes with methods. For example, you've seen the TryParse method that's part of the int type. You can use this method whether you declare your variable as type int or Int32, since they're the same. Microsoft encourages us to use the C# names (such as int), not the .NET names (such as Int32) when creating variables. Here's a list.

Java, however, takes a different approach. Java has several basic types called "primitive" types, including int, float, and boolean. They all start with lowercase letters. These are basic types, not classes, and as such they don't have any methods attached to them. Java then supplies what they called "wrapper classes" that are full classes with methods attached to them. These all start with uppercase letters. The wrapper type for int is called Integer. (Here's a list.) When you use Integer instead of int, you then get a large selection of useful methods. Integer, for example, has a method called toString for creating a string version of the number. This method is not available when you use the int type.

String, on the other hand, has no primitive type. It is a class and must always be capitalized.

Combining Strings

To combine strings, we can use the + operator just like with C#. Here's an example:

```
String first = "Doctor";
String last = "Dexian";
String full = first + " " + last;
System.out.println(full);
```

Getting Input from the User

To read input from the console in Java, you need to use a class called Scanner. But in order to use that class, you need to import it. We cover more on this later in this document, but this is similar to the "using" statements in C#. You need to add the following line at the top of your code:

```
import java.util.Scanner;
```

Then in your code you need to create an instance of the Scanner class. After that you can easily read from the console. Here's an example:

```
Scanner sc = new Scanner(System.in);
String name = sc.nextLine();
System.out.println("Welcome " + name + "!");
```

Notice that we're using the + sign to combine the strings together. We can also use a function called Format like so:

```
Scanner sc = new Scanner(System.in);
System.out.println("What is your first name? ");
String first = sc.nextLine();
System.out.println("What is your last name? ");
String last = sc.nextLine();
```

```
System.out.println(String.format("Welcome %s %s!", first, last));
```

Tip: C# has several nice ways to format strings, including string interpolation. Java has a more basic form called string formatting. Here's an excellent article on it.

You can also read in numbers. Here's an example:

```
Scanner sc = new Scanner(System.in);
System.out.println("What is your favorite number?");
int num = sc.nextInt();
System.out.println(String.format("Your favorite number times two is
%d.", num * 2));
```

Notice here we use sc.nextInt() rather than sc.nextLine(). We also have at our disposal nextBoolean(), nextFloat(), and nextDouble().

And here's an example of a while loop that asks the user for input. Here we ask the user to choose an option, and then we do an appropriate calculation and print it out.

```
Scanner sc = new Scanner(System.in);
boolean done = false;
int num = 1;
while (!done) {
    System.out.println("The current number is:");
    System.out.println(num);
    System.out.println("What would you like to do?");
    System.out.println("(1) Add 2");
    System.out.println("(2) Multiply by 2");
    System.out.println("(3) Multiply by 3");
    System.out.println("(4) Quit");
    int choice = sc.nextInt();
    if (choice == 1) {
        num = num + 2;
    }
    else if (choice == 2) {
        num = num * 2;
    }
    else if (choice == 3) {
```

```
        num = num * 3;
    }
    else if (choice == 4) {
        System.out.println("The final number is:");
        System.out.println(num);
        done = true;
    }
    else {
        System.out.println("That is not a valid choice. Please try
again.");
    }
}
```

WARNING: nextLine Gotcha!

Due to the mechanics of how nextLine works vs. nextInt, nextDouble, etc., you need to include an extra nextLine after a nextInt/Double/Etc before you use a nextLine. This technique clears out some extra space from the input buffer. *This should only be used when transitioning from nextInt/Double/Etc to nextLine.*

```
// This will not work. 💀 name will always be empty string ("")
System.out.println("How many games would you like to play?");
int gameCount = scnr.nextInt();
System.out.println("What is your name?");
String name = sc.nextLine();

// The following is correct 👍
System.out.println("How many games would you like to play?");
int gameCount = scnr.nextInt();
sc.nextLine(); // discard newline character
System.out.println("What is your name?");
String name = sc.nextLine();
```

Comparing Strings

In C# we have the benefit of easily comparing strings using the == operator. In Java, however, we can't use == in the same way. Instead we have to call equals() on one of the strings like so:

```
Scanner sc = new Scanner(System.in);
System.out.println("What is your name name? ");
String name = sc.nextLine();
if (name.equals("Doctor Dexian")) {
    System.out.println("Welcome Doctor Dexian!");
}
else {
    System.out.println("Greetings our friend, " + name);
}
System.out.println(name);
```

Tip: Be careful when you're comparing strings. Java does allow the == operator, but it doesn't actually compare the values stored in the strings. Not understanding this can easily result in bugs in your code.

(Optional) A deeper dive into comparing strings

Earlier in the course, we learned that C# can store variables in a shared memory area called the Heap, or a local memory area called the Stack. Strings are stored in the heap. That means when you create a list of numbers such as List<int> and store it in a variable, the list is actually in the heap. Then when you copy that list to another variable of the same type, both variables now point to the same list. Remember that strings also get stored in heap in the same way.

Java uses a similar approach. So when you create two strings in Java, and you want to try to compare them with the == operator, unlike C#, Java will see if they're the same string in memory; that is, it will see if the two string variables point to the same string stored in the heap. So even if you have two separate strings that happen to hold the

same sequence of letters, in Java, == will not figure out that they're the same sequence of letters, and will give you back a false.

In general, you'll want to always use equals() to compare two strings unless you really need to know if they're the same string object in memory.

Advanced Tip: You might try to test this out in Java with code like the following, and you might find that it doesn't seem to fit with what we just said:

```
String first = "Hello world!";
String second = "Hello world!";
System.out.println(first == second);
```

This will actually print out "true". The reason, however, is that the Java compiler is smart and while it was compiling this code, saw that the two strings were identical and therefore made only a single string and pointed both variables, first and second, to that one string.

Arrays

As with most languages, Java has arrays built into the language. It also has collections like in C# (see below), which like C# are considerably more powerful. But first let's see how to build some arrays and work with them. Here we'll create an array of integers:

```
int[] nums = new int[5];
nums[0] = 524;
nums[1] = 627;
nums[2] = 845;
nums[3] = 128;
nums[4] = 223;
for (int i = 0; i < nums.length; i++) {
    System.out.println(nums[i]);
}
```

Notice that we set the variable's type (on the left) with the basic type (in this case int) and then two brackets without anything inside them. On the right, we call new and then specify the size inside the two brackets.

In this case we're using a size of 5, which means the indexes, like in C# and many other languages, run from 0 up to and including 4.

You cannot change the size of an array after it's created. Further, if you try to access an index that's outside of the array, you'll get an exception when the program runs:

```
nums[5] = 8;
```

This will result in the following error message:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5

(The compiler doesn't check for such errors; the error only happens once the program is running.)

We can loop through arrays using both a regular for loop or the Java equivalent of a for-each loop. We just saw how to use the regular for loop. And now here's how you do the equivalent of a for-each loop in Java:

```
int[] nums = new int[5];
nums[0] = 524;
nums[1] = 627;
nums[2] = 845;
nums[3] = 128;
nums[4] = 223;
for (int num: nums) {
    System.out.println(num);
}
```

This is an easy way to loop through a string, but remember, like in C#, you don't have access to the index. If you need the index, you'll want to use the regular for loop.

We also have available a different way to initialize an array, similar to C#:

```
int[] nums = new int[] {
    524, 627, 845, 128, 223
};

for (int num: nums) {
    System.out.println(num);
}
```

Notice with this approach you don't put the size in the brackets after new. Java figures out the size based on how many elements you're initializing the array with.

Like in C#, arrays are mutable in the sense that you can replace any element:

```
int[] nums = new int[] {
    524, 627, 845, 128, 223
};

for (int num: nums) {
    System.out.println(num);
}

Scanner sc = new Scanner(System.in);
System.out.println("What would you like to change the third element
to?");
nums[2] = sc.nextInt();
for (int num: nums) {
    System.out.println(num);
}
```

And now we can look at how to use the for-each approach with a string. While you can't use a string itself in the for-each statement, you can ask the string for an array representation of itself, and then use that. Here we go:

```
String name = "Doctor Dexian";
for (char ch: name.toCharArray()) {
    System.out.println(ch);
}
```

Collections

Like C#, Java has collections that provide features that go far beyond what arrays give us. We won't cover them all, but rather just look at a couple. First is one called ArrayList. This is a basic list, and unlike arrays, it doesn't have a fixed size. The ArrayList also includes a lot of handy methods we can try out. Here we'll create a new list of strings, add some names to it, remove one name, and use a couple types of loops.

But first, we need to add an import statement at the top:

```
import java.util.ArrayList;
```

Now we're ready.

```
ArrayList<String> names = new ArrayList<>();
names.add("Emily");
names.add("Omar");
names.add("John");

// Now let's insert a name into the second position
names.add(1, "Sara");

// Let's loop using a regular for loop
for (int i = 0; i < names.size(); i++) {
    System.out.println(names.get(i));
}

// Let's remove one name. We can remove by name or by index.
names.remove("John");

// Now let's loop with a for-each
for (String name : names) {
    System.out.println(name);
}

// Tip: Create and fill an ArrayList in one line
ArrayList<Integer> nums = new ArrayList<>(Arrays.asList(1, 1, 2, 3));
```

Notice that like C#, we specify in angle brackets the type that we're putting into the list.

Notice that in order to get the size of the list, we call the Size() method. Also notice that to access the individual elements in the list, we can't use the bracket notation, unlike lists in C#. Instead we call the get function.

We also have a class called HashSet. This is similar to a list, except for two main differences:

- HashSets do not have duplicate elements. If you try to add the same element twice, it will still only appear once in the set.
- HashSets do not have a specific order to them.

Although HashSets don't have an order, we can still loop through them.

First add an import statement:

```
import java.util.HashSet;
```

Now we can use the HashSet.

```
HashSet<String> fruit = new HashSet<>();
fruit.add("Apple");
fruit.add("Cantaloup");
fruit.add("Banana");


// Now let's loop through them.
System.out.println("Here are our fruit!");
for (String item: fruit) {
    System.out.println(item);
}

// We'll add Apple again. Notice there's still only one.
fruit.add("Apple");
System.out.println("Here is the result of re-adding Apple:");
for (String item: fruit) {
    System.out.println(item);
```

```
}

// Let's remove Banana. Unlike list, we don't have indexes.
System.out.println("Now we've removed Banana.");
fruit.remove("Banana");
for (String item: fruit) {
    System.out.println(item);
}

// Let's try removing something that isn't present
fruit.remove("Strawberry");
System.out.println("We tried removing something that wasn't there:");
for (String item: fruit) {
    System.out.println(item);
}

// Let's see if Apple is in the set.
if (fruit.contains("Apple")) {
    System.out.println("This basket of fruit contains an Apple");
}

// Tip: Create and fill a HashSet in one line
HashSet<Integer> nums = new HashSet<>(Arrays.asList(1, 2, 3, 5));
```

Tip: If you're curious about the name HashSet, it's because there's a computer science technique for storing items in memory called a "hash table." Wikipedia has an article on it you can read; however, note that you do not need to know this to use a HashSet.

The code above should be rather straightforward. We don't have a regular for loop because sets don't have indexes. We call add() to add an item to a set; we can call remove() to remove an item; we can check if the set contains an item by calling contains(); and we can loop through the set using a for-each construct.

Generics and wrapper classes

When declaring a collection type, include the type of the collection elements in angle brackets (<>). This is called a "generic" type. C# has a very similar concept. There are two special considerations here:

1. When providing the variable type, you must specify the generic type. However, when instantiating the collection, you should not. The compiler may give you a warning. Just use empty angle brackets.

```
// WARNING: 💀 generic type not necessary with "new"
ArrayList<String> words = new ArrayList<String>();
// CORRECT: 👍 use empty brackets with "new"
ArrayList<String> words = new ArrayList<>();
```

2. Primitive (lowercase) types cannot be used as generic types. You must use the wrapper class (uppercase).

```
// ERROR: 💀 primitive type int is not allowed here
ArrayList<int> nums = new ArrayList<>();
// CORRECT: 👍 use wrapper class
ArrayList<Integer> nums = new ArrayList<>();
```

Collection interfaces and implementations

ArrayList and HashSet are two of the most common Java collections, but there are several more.

In Java, collections are categorized into several main types, each of which has an interface: java.util.List, java.util.Set, and java.util.Map.

List	Set	Map
• Ordered • Indexes / Indices	• In or out • No order • No duplicates	• Values accessed by keys • No duplicate keys • No order • Duplicate values are okay

Because these are interfaces, you cannot instantiate them directly.

```
// ERROR: 💀 interface List cannot be instantiated
List<String> words = new List<>();
// ERROR: 💀 interface Set cannot be instantiated
Set<String> tokens = new Set<>();
```

Instead, each interface has several pre-build implementation classes to choose from, each with the same capabilities but different performance characteristics. It's up to you to choose the implementation that's best for your situation, but most of the time you can safely go with the standard most common option. Here are some of the common implementations:

Interface	Implementation	Features
List	**ArrayList** (most common)	Efficient to access any index
	LinkedList	Efficient to add/remove/insert anywhere
Set	**HashSet** (most common)	Faster
	TreeSet	Always sorted
Map	**HashMap** (most common)	Faster
	TreeMap	Always sorted

With the interface/implementation distinction, you might see code like the following, where the variable has the interface type but the instantiation uses the concrete implementation type.

```
List<Character> letters = new ArrayList<>();
Map<String, Integer> tallies = new HashMap<>();
```

There are other cases, such as method parameters where the flexibility of using the interface instead of a specific implementation is important. This allows the method to accept any type of implementation.

```
public static List top(List<String> list, int limit) {
    if (list.size() <= limit) {
        return list;
    } else {
        return list.subList(0, limit);
    }
```

```
}
```

Style conventions

Capitalization

Java and C# both have “best practices” for how to name your variables, functions, and class names. Technically both compilers let you use any case you want; for example, you could call a class rectangle or Rectangle or RECTANGLE in either language. But the developers of the language gave us some conventions that we should abide by to help make our code more readable. Here is the list of conventions. But here’s a shorter list for Java:

- **Class names** should start with an uppercase letter before each word, and not use underscores. Examples include:
    - Rectangle
    - ProductPrice
- **Methods** should start with a lowercase letter. Then each subsequent word should be uppercase. Here are some examples:
    - calculate()
    - toString()
    - getArea()
- **Variables**, **fields**, and **method parameters** follow the same standard as methods. For example:
    - size
    - count
    - totalCount
    - currentAreaOfRectangle

C# in turn has a different set of standards, which your instructor might have discussed earlier. To review:

- **Class names** should be named just like with Java; start with a capital letter, and each word should start capitalized. Examples include:
    - Rectangle
    - ProductPrice

- **Methods** and **properties** should follow the same convention as classes. Examples include:
    - Calculate()
    - ToString()
    - GetArea()
    - Color { get; set; }
- **Variables**, **fields**, and **method parameters** should start with a lowercase letter, and then each subsequent word is capitalized. For example:
    - size
    - count
    - totalCount
    - currentAreaOfRectangle

Here's where you can read more about C# type name conventions.

Brackets

While C# code typically has the opening brackets of a code block on its own line, Java style typically puts the opening bracket at the end of the preceding line for a more compact look.

```
// C# Style
public bool IsItEven(int num)
{
    if (num % 2 == 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

```
// Java Style
public boolean isItEven(int num) {
    if (num % 2 == 0) {
        return true;
    } else {
```

```
        return false;
    }
}
```

Java 8 Steams

C# collections can be used with LINQ to conveniently perform common operations. The Java equivalent is something called Streams. This is often referred to as Java 8 Streams since they were introduced in Java version 8. These should not be confused with input/output streams, which are another feature of Java used for reading and writing files and other data sources.

Here's a tutorial if you want to learn more about Java 8 Streams.

Additional Resources

- Java API specification - Standard reference for Java types and collections. Use the search box in the upper right.
- Java 8 Streams Tutorial - from Baeldung