**Work Shop on Java SE 8 New Features**

**Update your java skills with Java SE8 New Features by AMR**

**Mode of Training: Hands on Class Room training**

**Duration: 3 hours**

**Fee: 200Rs**

**Prerequisites: Should familiar with java**

**Tools: Eclipse/MyEclipse**

**Material and exercise files will be forwarded to the your mail ids at the end of the session**

**Syllabus:**

1. Introduction
2. Getting Started
   1. An overview of Java SE 8 release
   2. Installing Java SE 8 on Windows
   3. Installing Java SE 8 Support for Eclipse
   4. Configuring Eclipse for Java SE 8
3. Using Lamda Expressions and Method Enhancements
   1. Understanding lamda expressions and method references
   2. Defining and instantiating a functional interface
   3. Traversing collections with lamda expressions
   4. Filtering collections with Predicate Interfaces
   5. Traversing collections with method references
   6. Implementing default methods in interfaces
   7. Implementing static methods in interfaces
4. Processing collections with streams
   1. Traversing collections with streams
   2. Creating streams from collections and arrays
   3. Aggregating stream values
5. Using New Date and Time
   1. Calculating Time spans with Instant and Duration
   2. Representing Date and Time values
   3. Formatting Date and Time values
   4. Supporting Time-Zone offsets
6. Using Javascript with the JVM
   1. Running java script from Java
   2. Writing Java script for Nashorn
7. Miscellaneous New Features
   1. Joining string values into delimited lists
   2. Searching text files with streams
8. About JAVA SE 8 Certifications
9. Java SE 8 Interview Questions

# Getting Started

**An overview of the Java SE 8 release**

Java SE 8 is a major release for the Java programming language and the Java virtual machine.It includes many changes. Some have gotten more coverage than others, but I'm going to talkabout both the major changes and a few of the minor ones. Probably the most attention has gone to Project Lambda, a set of new syntactical capabilities that let Java developers work as functional programmers. This includes lambda expressions, method references and a few other capabilities.

There is a new API for managing dates and times. Replacing the older classes. Those older classes are still in the Java Runtime, but as you build new applications, you might want to move to this new set of capabilities, which let you streamline your code and be a little more intuitive in how you program. There are new classes to manage local dates and times and time zones and for calculating differences between different times. The Stream API adds new tools for managingcollections including lists, maps, sets, and so on.

A stream allows you to deal with each item in a collection without having to write explicit looping code. It also lets you break your processing into multiple CPUs. So, for large, complex data sets you can see significant performance improvement. The Nashorn JavaScript engine is new to Java SE 8 too. This is a completely new JavaScript engine written from scratch that lets you code in JavaScript but lets you integrate Java classes and objects. In the chapter on Nashorn, I'll describe how to run Nashorn code from the command line.

But also how to write JavaScript in separate files, and then execute those files from your Java code. There are also enhancements to the concurrency framework, which lets you manage and accumulate values in multiple threads. There are lots of smaller changes as well. There are new tools for creating delimited lists in the string class and other new classes. There are tools for aggregating numbers including integers, longs, doubles, and so on. There are also tools for doing a better job of detecting null situations, and I'll describe all of these during the course.

And I'll describe how to work with files, using new convenience methods. So, when is Java SE 8 available? The answer is, now. It was released on March 18, 2014. For developers who use Java to build client site applications, the JavaFX rich internet application framework supports Java 8 now. And most of the Java enterprise edition vendors support Java 8 too. Whether you move to Java SE 8 right away depends on the kinds of project you're working on.

For many server and client site applications, it's available immediately, but Android developers beware, Java SE 8 syntax and APIs are not supported in Android at this point. It's only very recently that Android moved to some of the newest Java 7 syntax. And so, it might take some time before Android supports this newest syntax or the newest APIs. But for all other Java developers, it's worth taking a look at these new capabilities.

## Installing Java SE 8 on Windows

In order to run Java SE 8 on Microsoft Windows, first check which version you have. Java SE 8 is supported on Windows 8, 7, Vista, and XP. Specifically, you'll need these versions. For Windows 8 or 8.1, you'll need the desktop version of Windows. Windows RT is not supported.You can run Java SE 8 on any version of Windows 7, and on the most recent versions of Windows Vista and Windows XP. On Server based versions of Windows, you can run 2008 and the 64-bit version of 2012.

If you want to work on Java Applets you'll need a 64-bit browser, these can include Internet Explorer 7.0 and above, Firefox 3.6 and above, and Google Chrome which is supported on Windows, but not on Mac. You can download the Java Developer Kit for Java SE 8 from the URL java.oracle.com. That will take you to the current Java homepage. Click Java SE, under Top Downloads. Then click the Download link for Java 8.

On the Downloads page read through the license agreement, and if you agree with its terms, you can accept it, and then you'll be able to download the JDK for your version of Windows. I'm running 64-bit Windows, so I'll be using jdk.8.windows.x64. You can download it anywhere on your hard disk. I've already downloaded it to my desktop. I'll double-click, to start the installer. If you see a User Account Control dialog, click Yes to accept it.

Then follow through the prompts. I'll accept all of the options on this screen and note the location on my hard disk where the JDK will be installed. Then I'll click Next and the JDK installation starts. Just as with earlier versions of the JDK, there are two separate installations.The developer kit and the run time. When you come to this screen, you are installing the run time and you can once again the location, where it will be installed. I'll click Next. And that will complete the installation.

It might take a few minutes for all of the files to be copied into place, but when you come to this screen, the Java SE Development Kit has been successfully installed. I'll Close the installer.And the next step, is to make sure that all of my environment variable's, are pointing to the new location, of my JDK. You can do this through the Windows Control Panel. On Windows 8, you can open the Control Panel by moving the cursor to the lower left corner, and right-clicking, and choosing Control Panel.

On earlier versions of Windows, go through the Start menu. On the Control Panel menu, locate the environment variable settings by typing environment in the search box. Then click the link to, Edit the system environment variables. There should be a button labeled Environment Variables at the bottom of this dialog box, and then you can set two variables. First, look for a variable named JAVA\_HOME. This variable isn't required by all Java frameworks, and in fact you don't really need it for this course.

But some environments do need it, so I usually like to set it up. If you don't see it under the, system variables, click New. Type the, Variable Name as JAVA\_HOME in all uppercase. The Variable Value is the actual location of your JDK. To find that, go to File Explorer or on older versions of Windows, Windows Explorer. Then navigate to your Program Files folder. On my system that's C colon Program Files, and then double-click into Java.

On my system, I have two versions of the JDK, for Java 7 and Java 8, and that's fine. You can have as many versions on your system as you like. But we'll use these environment variables to point to the most recent version. I'll double click-into the JDK folder, and then click on the folder icon up here. That turns the location into a string, and then I'll copy that to the clipboard. Then I'll come back to this dialog box, where I'm setting up my variables, and I'll paste that string into place, for the variable value, and I'll click OK.

So that's my JAVA\_HOME variable. Also adjust your Path variable. This should already exist on your system and this points to folders that contain applications you want to be able to run from anywhere on your system. I'll double-click on to Path and then look through that variable, and see if you already have something for Java. On my system I'm already pointing to JDK 1.7, and i will adjust that value taking out the suffix, so it only says 1.7.0 and then I'll change it from 1.7 to 1.8.

Again, make sure that this exactly matches the actual folder location on your computer. I'll click OK, and click OK again, and click OK a third time, to exit the Control Panel. Now, I'm going to test and make sure that Java 8 is available. I'll do this through a command prompt. If you're working in Windows 8, you can once again move the cursor down to the lower left corner, right-click, and choose Command Prompt. If you're working on an older version of Windows, open the Start menu and type cmd into the search box, and that should take you to the same sort of command prompt.

In the command prompt, type java dash version, separated with a space. You should see your Java version is 1.8.0, or if you have a later update it might be 1.8.0, underscore and then some numeric value. Also check to make sure your compiler is available. With javac -version, and I get back javac 1.8.0. If you're working in a server environment, that's it you're finished.

But if you're going to be working on client side applications, you might also want to check the Java control panel. To get to that on Windows 8, go to the start screen and type Java. And that should show you an item labeled Java Mission Control. On older versions of Windows, you can locate this through the Start menu. Mission Control is a client-side application. The first time you run it after installing a new version of Java, you might see a firewall notice. If you do, click on Allow access.

If you're running Firewall or security software other than the built-in Windows software, your experience might vary. When you get to this screen, you can close the welcome screen. And if you're not running any client-side applications, you might see this dialog, saying that, no local JVMs were detected. But you can click OK. And as long as you're able to see that Java Mission Control has started then you'll know that your run time is healthy and you're ready to startprogramming and testing with Java SE 8 on Microsoft Windows.

**Installing Java SE 8 support for Eclipse**

Java SC8 is supported by all of the major Java development environments. Including Oracle's Netbeans, Intellij Idea, and Eclipse. For this course I'll be doing all of my demos in Eclipse, using Kepler, version 4.3.2. This version of Eclipse supports Java 7 out of the box. But it's very easy to add Java 8 support. If you don't already have Eclipse installed, you can get it from here at www.eclipse.org/downloads.

I won't go through the installation process from scratch. But it's really simple. You'll get an archive file, and you just extract it anywhere on your hard disk. I'm using the Eclipse IDE for Java developer's distribution. And I've already extracted it to my desktop in a folder named Eclipse. I'll start Eclipse up from here. I recommend using a brand new work space for your Java 8 work.That will allow you to configure that workspace for Java 8, and not to worry about conflicting with earlier Java projects.

I'll name my workspace, Java 8WS. And put it under my home folder. Now, before you do any Java 8 coding. You will need to install Java 8 support for Eclipse. You can find information about this on this webpage at the Eclipse Wiki. The URL is wiki.eclipse.org/JDT/Eclipse\_Java\_8\_Support\_For\_Kepler with appropriate underscores where needed. Right toward the top of this page there is a URL that you can use in Eclipse to download the Java 8 support.

I'll select that and copy it to the clipboard. Then I'll go back to Eclipse. From the menu I'll select Help > Install New Software. Then I'll paste that URL into place and click add. In the add repository dialogue, you can name this location anything you want. I'll just name it Java 8 Support. And I'll click okay, and then wait a few moments, while Eclipse interrogates that URL.There are two components that are available. You definitely need the first labelled Eclipse JDT.

That's the java development tools for all java development. You probably don't need the secondcomponent, that's for building Eclipse plugins. So I leave that unchecked. To speed up the process a little bit, you can uncheck the option to Contact all update sites during the installation.Then click Next. Check to make sure you're installing what you want, and then click Next again.Review the License agreement and if you accept its terms. Click the appropriate radio button and click finish. It might take a few minutes to download and copy all the files into place for Java SE support.

But when you're done, you'll be prompted to restart Eclipse to allow the changes to take effect.I'll click Yes, and then wait a moment until Eclipse restarts. I'll select the same workspace as before, and then when Eclipse has completely started up I'll be ready to configure it and test Java 8 support.

**Configuring Eclipse for Java SE 8**

Once you've installed Java SE 8 support for Eclipse, you're ready to configure your Eclipse workspace and test Java 8 on your system. Go to the Menu and go to your Preferences dialogue. On Mac you should go to your Eclipse menu and chose Preferences. And on Windows go to the Window menu and chose Preferences. Here are two settings to check. Open the category labeled Java, and then click on Compiler. Your compiler compliance level should be set to 1.8. If it isn't set that way, set it now, and if you don't see 1.8 as an available option, that means that your Java SE 8 support was not correctly installed.

Go back to the previous movie in this chapter. And walk through that process again. I see it there, so I'll select it. You should also check to make sure that you have a Java 8 runtime installed. Click on, Installed JREs. And, if everything works correctly, and you've created a brand new workspace after installing Java 8, you should see the Java 8 runtime listed. The location on my system is under C:\Program Files\java\jre8.

If you don't see this on your system you can set up the JRE from here by clicking the Add button, selecting Standard VM and then filling in all of the required information. Once you have the Java 8 runtime listed, just check it to make sure that it's your default. And then click OK to close the Preferences dialogue. Now you're ready to test Java 8. I've provided a project for this purpose in the exercise files. I'll import it into Eclipse. I'll choose File > Import.

Under General I'll choose Existing Projects into Workspace. Then I'll choose Select archive file and browse. Starting from your exercise files folder go to 01 getting started and import the file testjava8.zip. Follow the rest of the prompts to complete the import process. This project has a package called org.examplejava8, and under that, two files. The first is called test interface.java.It defines a simple interface with a single public method named test.

Most of this code would work on Java 7, but there's an annotation here that was added in Java SE 8, FunctionalInterface. If your Eclipse environment isn't correctly configured for Java 8, this annotation will cause an error because it won't be recognized as valid Java code. I don't see an error, so that's a good sign that Eclipse is working as I hoped. Next I'll open this class definition named Testjava8.java. This is a class with a main method so I can run it as a console application and it has a critical line of new Java 8 syntax.

It's creating an instance of that functional interface I just showed you using a lambda expression, a style of syntax that didn't exist in Java prior to Java 8. I'll explain what this syntax is doing very early in the course. But all you need to know right now is that if this code isn'tcausing any errors, then once again, Eclipse is recognizing it as valid Java syntax. I'm creating an instance of that interface and then calling that interface's test method. And so, I'll run the code.

I'll click the Run button on my tool bar and in my console I see a successful result. I've created an object which is an instance of that interface using a lambda expression. And I've called its method and it's outputting a string to the console. So, if this is all working, you're in great shape.You're ready to get started programming with Java SE 8 in Eclipse. If you had any problems along the way, go back to earlier in the chapter and walk through the steps. One step at a time.

If you were using an older version of Eclipse and tried to upgrade it, you might try downloading a fresh copy of Eclipse. And if, by any chance, you were using a prerelease version of Java SE 8, some release from the very long public beta period, then I recommend that you uninstall that and install the release version. By working with the most recent versions of all of the software, you'll ensure that you can follow along with all of the demonstrations in this course.

# 2. Using Lambda Expressions and Method references

## Understanding lambda expressions and method references

Perhaps the best known new feature of Java SE 8 is called Project Lambda, an effort to bring Java into the world of functional programming. In computer science terminology, a Lambda is an anonymous function. That is, a function without a name. In Java, all functions are members of classes, and are referred to as methods. To create a method, you need to define the class of which it's a member. A lambda expression in Java SE 8 lets you define a class and a single method with very concise syntax implementing an interface that has a single abstract method.

It lets developers simplify and shorten their code. Making it more readable and maintainable.Let's take a look at a few code snippets. Prior to Java SE 8, if you wanted to create a thread, you'd first define a class that implements the runnable interface. This is an interface that has a single abstract method named Run that accepts no arguments. You might define the class in its own code file. A file named by MyRunnable.java. And you might name the class, MyRunnable, as I've done here. And then you'd implement the single abstract method.

In this example, my implementation outputs a literal string to the console. You would then take that object, and pass it to an instance of the thread class. I'm instantiating my runnable as an object named r. Passing it to the thread's constructor and calling the thread's start method. My code will now run in its own thread and its own memory space. Prior to Java SE 8, you could improve on this code a bit, instead of declaring your class in a separate file, you might declare it as single use class, known as an inner class, local to the method in which it's used.

So now, I'm once again creating an object named r, but I'm calling the interface's constructor method directly. And once again, implementing it's single abstract method. Then I'm passing the object to the thread's constructor. And you can make it even more concise, by declaring the class as an anonymous class, so named because it's never given a name. I'm instantiating the runable interface and immediately passing it to the thread constructor. I'm still implementing the run method and I'm still calling the thread's start method.

In Java SE 8 you can re-factor this code to significantly reduce it and make it a lot more readable. The lambda version might look like this. I'm declaring an object with a type of runnable but now I'm using a single line of code to declare the single abstract method implementation and then once again I'm passing the object to the Thread's constructor. You are still implementing the runnable interface and calling it's run method but you're doing it with a lot less code. Let's break this down to understand what's happening.

Here is an important quote from an early specs document about Project Lambda. Lambda expressions can only appear in places where they will be assigned to a variable whose type is a functional interface. You can find this quote and a lot more details about lambda expressions at this URL. A functional interface is an interface that has only a single custom abstract method.That is, one that isn't inherited from the object class. Java has a lot of these interfaces such as Runnable, Comparable, and many others.

Prior to Java 8 they were known as Single Abstract Method or SAM interfaces. In Java 8 we now call them functional interfaces. This lambda expression is returning an implementation of the runnable interface, it has two parts separated by a new bit of syntax called the arrow token or the Lambda operator. The first part of the lambda expression, before the arrow token, is the signature of the method you're implementing.

In this example, it's a no arguments method so it's represented just by parentheses. But if I'm implementing a method that accepts arguments, I would simply give the arguments names. I don't have to declare their types. Because the interface has only a single abstract method, the data types are already known. And one of the goals of a lambda expression is to eliminate unnecessary syntax. The second part of the expression, after the arrow token, is the implementation of the single method's body.

If it's just a single line of code, as with this example, you don't need anything else. To implement a method body with multiple statements, wrap them in braces. Lambda Expressions can reduce the amount of code you need to write and the number of custom classes you have to create and maintain. If you're implementing an interface for one-time use, it doesn't always makesense to create yet another code file or yet another named class. A Lambda Expression can define an anonymous implementation for one time use and significantly streamline your code.

In this chapter's demonstrations, I'll show you a few different uses of Lambda Expressions.They all have in common the implementation of functional interfaces. I'll explain how the compiler is inferring information from code, such as specific types of variables and what's really happening in the background.

## Defining and instantiating a functional interface

To get started learning about Lambda expressions, I'll create a brand new functional interface.An interface with a single abstract method, and then I'll implement that interface with the Lambda expression. I'm working in a project named Functional interface that you can import from the courses exercise files. The project has a single default package named column.example.javase8. I'll create a brand new interface. I'll right-click on that default package and select New > Interface.

I'll actually put the interface into its own sub-package ending with .interfaces. And I'll name the interface, SimpleInterface, and I'll click Finish. In order to implement an interface with a lambda expression, it must have a single abstract method. I'll declare a public method that returns void, and I'll name it doSomething. It won't accept any arguments. And that's all you need to do tomake an interface that's useable with Lambda expressions. If you want, you can use a new annotation, that's added to Java SE 8, named Functional Interface.

I'll type the @ character, and FUNC and press Ctrl+Space. And eclipse auto completes that annotation. I'll save the change, and now I'm ready to create a new class. I'll go back to my default package and right-click and create a brand new class, and I'll name this Usesimpleinterface. I'll be running this class as an application. So, I'll select the option to create a main method, and click Finish. I'll get rid of the auto generated comment within the main method, and I'll add a little bit of code here.

My goal is to instantiate my functional interface, so I'll start with the type of the interface, SimpleInterface. And as I auto complete it, the import statement is added for it at the top. I'll name the object that I'm creating simply obj. Now I'm going to double-click the tab here to get a wider editor, and now I'm ready for my lambda expression. The simple interface has a single abstract method named do something, which accepts no arguments. To represent the signature of that method, I'll use a pair of parentheses.

Then I'll add in the arrow token. That's the lambda operator. And then I'll implement my method with a single line of code. I'll type sysout, and press Ctrl+Space. And that's an Eclipse code template that expands to System.out.printline. And I'll pass in a literal string of, say something.And now I've declared my object, which is an implementation of simple interface. And I can call the object's doSomething method. I'll type obj, and then a dot, and then auto complete the doSomething call.

I'll save my change, and run the code by clicking the Run button on the toolbar, and there's the result. With a simple lambda expression, I've created this object, which is an implementation of the interface, I've implemented the single abstract method with just a simple line of code, and I've been able to call the object's method. So that's what the code can look like when you'reworking with a single abstract method that doesn't accept any arguments. Let's take a look at what it looks like with arguments. I'll go back to my Package explorer, and I'm going to copy and paste my simple interface file.

I'll select it and press Ctrl+C and then Ctrl+V. And I'll name this new version interfacewithargs.I'll close my open editors and open the new interface. I'll change the signature of this method sothat instead of no arguments it receives two arguments. It'll receive two integer arguments, and within the interface those arguments will be named, value1 and value2.

I'll save and close this new interface, and then I'll go back to my default package, and create another new class. I'll name this class, useinterfacewithargs. And once again because I'll be calling it as an application, I'll add a main method. Within the main method, I'll declare an object data typed as interfacedwithargs. I'll press Ctrl+Space to add the import. And I'll name this object obj, just like I did before. After the equals assignment operator I'll put in a pair of parenthesis, but I need to mimic the signature of this method from the interface, which expects two arguments.

They were named value1 and value2 in the interface, but you can name them anything here. I'll name them v1 and v2. I don't need to put in int before the argument names; that information is already known. Then I'll put in the arrow token, or the lambda operator, and I'm ready to implement the method. This time I'll wrap the method implementation with a pair of braces, followed by a semi-colon to end that whole statement. And then I can add as many lines of codeas I want between the braces to implement the method.

Within the method, I'll declare a local variable typed as int, I'll name it result, and I'll get its value by multiplying v1 and v2. Then I'll use system output, and I'll output a literal string of the result is. And then I'll concatenate the result. So now I've declared the object and I'm ready to use it.And so I'll use the object variable obj, then I'll call the method doSomething and I'll pass in literal values, integer values of ten and five.

I'll check my code for any errors or warnings, and I don't see any, and now I'm ready to test my code. I'll click the Run button, and I see that the result is 50. Now if want to change the calculation, I can do it really easily. I'll just go back to the implementation of the interface. And I might change this from an asterisk for multiplication, to a plus for addition, and I'll Save and Run and now the result is 15. And finally, I'll show that you can also re-factor your code. I'll go back to the interface file, InterfaceWithArgs.

And I'll double-click on the doSomething method. And I'm going to re-factor it. I'll rename it as calculate. I'll right-click and choose Refactor > Rename, I'll name it calculatehere. I'll press Enter and then save my file. And go back to the class and show that the method name has been changed here as well. Eclipse and all of the other Java environments that support Java SE 8 are fully aware of this new Lambda expression syntax, and they'll support refactoring to the same extended that they do in other parts of the Java programing language.

When you create your own functional interfaces, you have complete control over their names and their method arguments. But again, you have to follow that rule that the interface can only have one abstract method. And then that interface and its single abstract method can be implemented with a lambda expression.

**Using built-in functional interfaces with lambdas**

I've previously described how to use a lambda expression to implement an interface that you've created yourself. Now, I'll show lambda expressions with built in interfaces. Interfaces that are a part of the Java runtime. I'll use two examples. I'm working in a project called BuiltInInterfaces,that's a part of the exercise files. And I'll start with this class. UseRunnable. In this class, I'm implementing the Runnable interface. This is an interface that's a part of the multithreaded architecture of Java.

In order to create a thread, you create an instance of the Runnable interface which has a single abstract method named run. I have two instances of the interface here, and I'm passing them to two instances of the thread class and calling the start method. I'll run the code a few times.Because these threads are running simultaneously, the question of which ends first is arbitrary.It depends on internals of the Java runtime. The first time I run it, thread two might end first.The second time thread one and so on.

But my focus here is on how you code, not in how it operates. I'm going to show how to uselambda expressions to replace these inner classes. I'll comment out the code that's declaring the two objects. And then I'll place the cursor down here and re-declare them and do the implementation with lambdas. I'll start with Runnable and I'll name this one r1, and then, after the assignment operator I'll put in a pair of parentheses. This matches the pair of parentheses with no arguments that's declared in the run method in the more verbose version.

Then I'll add the arrow token and then I'll copy the one functional line of code that outputs to the console and I'll paste it right here. And that's all I need. Now I'll do duplicate that line of code and for the second one I'll change it to r2 and change the output accordingly. I'll save the change and run the code. The behavior is exactly the same, using lambda expression in this context doesn't give you any performance benefit.

It's a simple way of reducing the amount of code you have to write. If you compare the two versions, the one with the inner classes took about ten lines of code. And the one with lambdas took two. It's more readable, and it's more maintainable. If you need to expand any of these methods to multiple lines, wrap them with braces. For example, I want to add a call to the sleepmethod to cause the first thread to pause for a moment. So, I'll place a starting brace after the token and when I press Enter or Return, Eclipse adds the ending brace for me.

And then I'll add the semi colon at the end of the brace to complete the statement. Now, I can add any code I want between the braces. I'll call thread.sleep. Now, when I press the period, I see this Problems During Content Assist dialog. You might see this a few times as you code using the new version of the Java Developer tools. It's a bit of a bug. And you can disable this feature as described in the dialogue, but I just ignore it. I'll click OK, and then I'll type thebeginning of the method I want to use, sleep.

And I'll use the version of the method that accepts a single long value. And I'll pass in a value of 1,000. That means, pause this thread for 1,000 milliseconds, or one second. I see an error token, over on the left. And when I float the mouse over it, it shows me there's an un-handled exception. So, I'll add a tri-catch block, around the call to the sleep method. I'll press Ctrl+1 for a quick fix. Or Cmd+1 on Mac. And choose surround with tri catch.

I'll get rid of this automatically generated comment. And then I'll save the changes and I'll once again run the code. This time, I see clearly that the second thread completes before the first.Because the first thread now has that one second pause. I'll click it again and see exactly the same result. So, lambda expressions can be used either with single lines of code or with complete code blocks. And you can use them effectively with the built-in interfaces with either single lines of code or with complete code blocks.

Let's look at another example. I'll close this file and open the file UseComparator.Java. In this Java class, I've created an array list of strings. Some are upper case, and some are lower case.Then, at lines 19 to 24, I'm calling the collections classes' static sort method. That sorts the array collection, and it does it with case sensitivity by default. Then I'm looping through the collection and outputting the strings to the console.

In the next bit of code, I'm using a comparator. The comparator is another functional interface inJava which has a single abstract method. This method is the compare method. It accepts two arguments. The type of the arguments depends on the generic declaration here. If you say that the comparator is for strings then the arguments will be strings. This is an instance of an anonymous inner class. I'm not naming the object, I'm declaring it and passing it into the sort method, all in one bit of code.

The compare method calls the string object's compare to ignore case method and passes in the second string object. It returns an integer. The integer value can be either negative 1, 0, or positive 1, depending on the results of the comparison. And that value is used by the sort method to figure out how to sort the values. Because I'm explicitly using the compareToIgnoreCase method, this will be a case insensitive sort. Here's the result of the current code.

The case sensitive sort, shown at the beginning, sorts all of the uppercase values to the top.And the lowercase values to the bottom. The case insensitive sort, done with the comparator object, does an accurate case insensitive sort. So, everything is purely alphabetical. Now, our goal is to learn about lambda expressions and this bit of code would work fine in earlier versions of Java too, but let's see how it might look with the lambda expression. I'll place the cursor before the call to the Sort method, the version that takes two arguments.

The collection and the comparator. Now I'll code up a comparator using a lambda expression.I'll choose the comparator interface and I'll set its generic type to string and I'll name it comp.Now, because the compare method takes two arguments, I need to pass those arguments in here. I'll pass them in, and name them str1, and str2. Then I'll add the arrow token, or lambda operator and a pair of braces. I'll add the semicolon after the second brace.

And now I'll take this bit of code, that's the code that's providing the functionality. And cut and paste it and place it in the lambda expression. Then, I'll come down here and I'll remove the code that's declaring the anonymous object. And I'll replace it with the comp object that I just declared with the lambda. And there's the refactored result. The functionality is exactly the same. When I use the second sort method and the comparator object it results in a caseinsensitive, purely alphabetical sort operation but the code is simpler, more concise and easier to read.

As before, it doesn't provide you any performance benefit. The underlying functionality is exactly the same. Whether you declare your own classes, use inner or anonymous inner classes, or lambda expressions, is completely up to you.

## Traversing collections with lambda expressions

In Java SE 8, you can use lambda expressions to traverse a collection of items. Examples of collections include lists, maps, and sets. All of these data types implement an interface called iterable. To understand the code I'm about to show you, let's start in the documentation. I'm working a project called TraverseCollection. Which has a class called main.java. I'll click into the class name ArrayList and then go to Dynamic Help.

And I'll open the Java docs for this class. The ArrayList class implemented a number of interfaces. This is the one I'm interested in called iterable. This interface was added in Java 5.And it has a method called iterater. That returns an instance of an iterater interface that you can use then to loop through the contents of a collection. But in Java 8 there were a couple of new methods. One called forEach, and one called spliterator. I'm going to focus on the forEach method.

It accepts an instance of a functional interface, called consumer. The consumer interface, has a single abstract method, named accept. And by implementing this interface and its single abstract method, you can add code that operates on an item in the collection. So, let's go back to the code. In the main class, I'm traversing this collection, this array list of strings, twice. In the first version, starting at line 23. I'm using a forEach loop, it's a simple bit of code, it create an str string variable for each item in the list and then executes whatever code is needed.

At line 36, I'm using another approach, calling the iterator method of the collection. Getting an iterator object and then looping with a while code block, calling the iterators has next method.Now both of these bits of code have worked fine going all the way back to Java 5, but I'll show you what this might look like with a lambda expression and the forEach method. In Java 8. I'll start with this for method.

I'll make a little bit of space here, and I'll comment out this code. I won't need that anymore. And then I'll reference my strings collection and call the new forEach method. And again it's going to receive an instance of the Consumer interface. Because Consumer is a functional interface, I can use a lambda expression. The Consumer interface's accept method requires a single argument of the appropriate data type. Because strings is declared as a list of strings, I have to pass in a string as the argument.

I'll pass it in with a name of str. Then I'll add my lambda operator, the arrow token. And I'll add my system output here, and pass in str. And I'll clean up the code a bit. And now this single line of code replaces the three lines of the forEach loop. I'll make a copy of this code and movedown here to the version that's using an iterater. And I'll select those lines of code and comment them out. And paste in my lambda expression version. So when you replace an iterator, you're replacing four lines of code with a single line.

Now save and run the code and see that it behaves exactly the same as it did before, but just with less code. So this is just another option for traversing collections. You can use forEach loops. You can use iterator objects and now you can use the forEach method with a lambda expression.

## Filtering collections with predicate interfaces

In addition to the new lambda syntax, Java SE8 adds a number of new functional interfaces.One of the most useful is called the Predicate Interface. An interface that has a single boolean method named Test, that you can use to wrap up your conditional processing, and make conditional code a lot cleaner. I'll show you how to use the predicate interface in this project named Filter Collection. Starting in the class named predicate inner class, I've declared a list of person objects.

Person is a custom class that's also a part of this project. It has two private fields. A string and an integer. Appropriate setters and getters. A constructor method that let's you easily create instances of the clas. And down toward the bottom an override of the to string method that outputs the name and age of the current person. Now my goal is to filter this list and only display certain people. You can use the predicate interface either with inner class syntax or with lambda syntax.

I'll start with inner class syntax. I'll place the cursor after the code that's adding people to the list.And I'll declare an instance of the predicate interface. I'll type the name of the interface and press Ctrl+Space. And choose the interface from the list. It's a member of the package Java.util.function, and you'll find a lot of other new functional interfaces there as well. This interface needs a generic type declaration. I'll set that to the name of my custom class, person.And I'll name the predicate, pred, P-R-E-D.

Now, for inner class syntax, I'll start with the new keyword, and I'll press Ctrl+Space and choose the constructor for the predicate interface. When I choose that, Eclipse automaticallyimplements the single abstract method, Test. Because I declared my predicate with a generic type of person, the test method accepts a single argument of that type as well. Now, I'm going to implement the method. I'm going to change the name of the person object being passed in to just P.

I'll get rid of that comment. And then I'll set the return statement to use a very simple criteria. I'll add a set of parentheses to wrap my condition. And I'll set the condition to P.getAge is greater than or equal to 65. So now my predicate object encapsulates my test, and the test method can be used to determine whether I want to process an object from my collection. The next step is to traverse the collection. And you can do this in a number of ways with a forEach loop, an iterator object, or with the new forEach method.

I'll use the classic forEach loop. I'll type foreach and press Ctrl+Space, and choose the foreach code template. Within the for loop, I'll use an if statement. And I'll set the condition to use the pred object that I just declared. I'll call pred.test. I'll pass in the person object that I'm declaring in foreach loop. And then if the condition is true, I'll use system output and I'll output the results of calling the toString method of the person object.

I'll save and run that code. And I only see people who are older than 65. Now, if I want to change my condition, I can change the predicate object and rerun the code and it'll work fine.But our goal here is to make the code as concise and readable as possible. And for that, you might decide instead to use a lambda expression to implement this predicate interface. So let's go back to the code, and I'll close this version, predicate inner class, and open this class, predicate lambda.

Now I'm going to do very similar processing. But I'll declare my predicate object this time using a lambda expression. Once again, I'll type the name of the interface. I'll press Ctrl+Space and choose it from the list, and set its generic type. And I'll name this predicate object predOlder.Now I'll implement the interface using a lambda expression. I'll start with the signature of the method I'm implementing. That's the Test method. And because I declared predicate with a generic type of person, the argument is going to be an instance of person.

I'll just name it P. Then I'll add the arrow token, and then I'll implement my method with a simple conditional expression. p.getAge greater than equal to 65. And that's all I need. This single line of code replaced five or six lines of code in the inner class version. Next, I'll loop through the list and use the test method of the predicate object. This time, I'll use the forEach method.

I showed how to use this method in a previous movie. It's a new method that was added in Java SE8. In the forEach method, I'll start with the name of the object I'm passing in as an argument.This time I won't wrap it inside parentheses just to show you a difference in syntax. Then I'll add the arrow token and a pair of braces. Within the braces I'll use an if statement. And I'll pass in my condition, which again the predicate's test method. I'll use predOlder.test, and I'll pass in the person object.

And if the condition is true, I'll use system output. And I'll output the person object. This time I'll refactor the code and not explicitly call the toString method. The Java run time will figure out that needs to call it for me. I'll save and run the code. And there is the result. Once again, I'm only displaying people whose ages are greater than or equal to 65. But what if I want to deal with more than one possible condition? Well this is the great thing about lambda syntax and the predicate interface.

You can make as many predicate objects as you want, each representing a different condition.So I'm going to duplicate this line of code, and I'll change the name of the new one to predYounger. And I'll change its condition to less than or equal to 40. To make it really easy to pass in which ever predicate I want, I'll take this code that's traversing the collection, and put it into its own separate method.

I'll select it, right-click, choose Refactor, and then Extract Method. And I'll give my new method a name of displayPeople. Now, the method doesn't know which predicate it's going to get, so I'm going to refactor the name of the predicate object in the method to just pred. And now, my existing code is passing in predOlder, so I'll run that. And I get Mike who's older than 65. And then i'll change the predicate object that i'm passing in to predYounger, and i'll run the code again.

And this time i get Mary who's younger than 40. So this is how you can use the new predicatetinerface and lambda expressions to encapsulate your conditions in individual objects. And then pass those objects into your own methods for processing. I won't cover a lot of the other new functional interfaces in Java SE8 in detail, but I encourage you to investigate the package that the predicate interface is a part of. Java.util.function. You'll find a lot of new useful interfaces there.

And because they're all functional interfaces, they can all be implemented with lambda expressions.

## Traversing collections with method references

In addition to Lambda expressions, Java SE 8's Project Lambda also adds method references to the language. A method reference gives you a way of naming a method you want to call, instead of calling it directly. And just like Lambda expressions, the goal is to make your code more concise and more readable. I'll demonstrate this in this project named MethodReferences,starting in a class called static method reference. You can use method references on four kinds of methods.

Static methods of any class, instance methods of a particular object, instance methods of an arbitrary object, in which case you would refer to it just like it were a static method, and references to constructor methods. I'll start with static methods. In this code, I have a list of instances of the person class, and my goal is to sort the data by the people's age. I'm going to create a method that does a comparison of two instances of the class. Now you could put this method anywhere you like, and developers will differ about the best place to put it, but I'm going to create the method as a static method of my person class.

I'll open the class and I'll place this new method down below the override of the toString method. I'll declare it as public and static, and set its return data type to int. I'll name the method compareAges. And the method will accept references to two instances of this class. I'll name the first one p1 and the second one p2. This method will know that the goal is to compare the ages of the two people. And just like the compare to method of the string class, and other compare to methods in Java, this will return a zero if the two values match, and then a negative 1 or a 1 to indicate that one is younger or older than the other.

I don't need to worry about those details which value to return because that will be handled by the integer classes compare to method. The code will look like this. I'll create an integer object that I'll name age1, and I'll get its value by calling p1.getAge. Then I'll return the value of the following expression. Age1.compareTo. And I'll pass in the second person's getAge method. So now I'm comparing two integers, and returning an integer indicating whether there is equality, or whether one is greater than the other.

I'll save that change, and now I'll go to my class static method reference. To sort the data, I once again use the collections class and its sort method. I'll type the name of the class and press control space to make sure its been imported above. Then I'll call the sort method. I'll use the second version of the method. That's expecting the collection and a comparator object. The collection is people. And for the comparator object, I'll use a method reference. It'll look like this, I'm referring to a static method, so i'll start with the type, the class that contains the static method.

Then I'll put in the double colon operator, that's how you separate the type or the object from the name of the method that you're calling. And then I'll pass in the name of the method, compareAges. Now here's why this is working. This method, the sort method, is expecting an instance of the comparator interface. That interface has a single abstract method, which expects two values. I'm calling a method that expects two values. And returning a data type that can be used by the sort method.

Now that the data has been sorted, I'll traverse the collection, and output the values. As I've done before I'll use the new forEach method. I'll pass in the current person as p. And then use system output to output the person. I'll save the change, and run the code, and there's the result. The data has been sorted by age. So that's a static method reference. If you prefer, you can use method references with instance methods. To do this, I'm going to go down here and select this code, the compareAges method that I just created in the person class, and I'll copy it to the clipboard.

And now I'll go to instance method reference. And i'm going to paste this method as a method of this class, and take away the static modifier. Now compareAges is an instance method of instance method reference. In this class, in the main method, I'm creating an instance of the current class, and then calling it sortData method. By the time I get down here, where I have created the list of people, all of the data and the methods are instance members, rather than static members.

I'll use the same sort of code that I did previously, when I used a static method reference, but there will be one critical change. Once again, I'll call Collections.sort. I'll choose the second version of the sort method. I'll pass in the people collection. And now for my method reference, I'll start with my object reference. If it's a named object, I can provide the name, but because I'mdealing with the current instance of the class already, I'll start with this. Then I'll put in the double colon, and I'll pass in the name of the method.

And just as I did before, I'll use the forEach method to output the person's information to the screen. And I'll save and run, and there's the result. So method references are a simple way of making your code incredibly concise. I will warn you that not all integrated development environments have caught up with this syntax. For example, in Eclipse, if I try to autocomplete the compareAges method, it won't work. I'm assuming that this sort of thing will be improved over time as the development environments catch up with the new capabilities in Java SE 8.

**Implementing default methods in interfaces**

Prior to Java SE 8, interfaces could contain abstract methods and constant declarations, but you couldn't provide fully implemented methods that would be inheritable. I'm working in a project called DefaultMethods. And in this application, I have an interface named PersonInterface.Java. It has four abstract methods, in interfaces all abstract methods are soon to be public, so I haven't included the public keyword and these are basic getters and setters.

Then I've a class named Person and this class has the setters and the getters implemented.And a constructor method that makes it easy to instantiate the class. And then I have a main class called use default method. In this code, I'm filtering a list of people using a predicate object, and then, in the static method, display people at lines 30 and 31. I'm putting together a string named info and outputting it to the console. So, I'm going to refactor this code using a new capability of Java SE 8 that lets me add something called a default method to an interface.

When you add a default method to an interface, you can add its full implementation. And then, any class that implements that interface will inherit that method and you can call it itself or themethod will be callable from anywhere else in the application because, just like the abstract methods, it will be public. I'll move the cursor below the abstract methods. And I'll start my method signature with the new keyword, default. The rest of the method will look exactly thesame, as if I were implementing it in a class.

I'll start with the return type, then the name of the method. Next, I'll add the code, and it'll be a return statement that can catonate the values of the name and the age. Now, because this is an interface, I can't refer to private fields. You can't do that. So, I'm just going to refer to the abstract methods, which I know will be implemented by the class itself. I'll call getName, I'll concatenate a space and an opening parentheses. Then I'll concatenate getAge and a closing parenthesis.

I'll save that change and now that method is available to every class that implements the interface. I won't make any changes to the person class. It already has that method. Then I'll go over here to my main class, use default method, and I'll change this code. I no longer need to create the string named Info, that's going to be done by the new method that the Person class has inherited. So, I'll delete that line of code. And I'll replace the reference to the info variable with a call to the getPerson info method.

But I'll call it as a member of the person object that I'm working with right now. I'll save the change and run the code. And there's the result. I'm successfully calling the getPerson info method to get a string, a concatenation of the values of the name and age, and then I'm outputting it to the console using the code in my main class. By using default methods, you cansometimes eliminate a whole layer of inheritance. Some developers, for example, in earlier versions of Java, might have created an interface, then a base class that implemented the interface and then a subclass that they would actually use in their code.

With this new capability, you might not need the base class at all and instead, can go directly to implementing the subclasses, inheriting the default methods directly from the interfaces.

## Implementing static methods in interfaces

I previously described how to add default methods to interfaces, that are fully implemented and are inherited by implementing classes. In Java 8, you can also add fully implemented static methods to interfaces. Just as with default methods, the goal is to let you eliminate inheritance layers and simplify your applications. I'm working in a project called StaticMethods. Just as in the earlier project, the one for default methods, the main class, which is called UseStaticMethod here, has code that gets the name and age from a person object.

It's at line 30 in this class. My goal is to take this code and move it to a static method, but instead of adding it to a base class or other concrete class, I'm going to add it to an interface. Something I wasn't able to do in earlier versions of Java. I'll take this code and copy it to the clipboard. Then, I'll open the interface, named personinterface.java. Just as before, I'm starting with a set of abstract method declarations. Now, I'll place the cursor after those abstract method declarations and I'll start with the keyword static.

Just as with the default method and the abstract methods, this will be a public method automatically. I don't need to declare it. It'll return a string and I'll name it getPersonInfo. Now because this is a static method, it can't refer to the instance methods declared above. So, I'm going to pass in an instance of the Person object. Then I'll provide the return keyword and I'llpaste in the code from before and I'll clean this up so that now I'm returning the name, the opening parenthesis, the age and the closing parenthesis.

I need to add an import statement for the Person class, so I'll place the cursor there and use a quick fix and import the class. And now this static method is available from anywhere in the application. Just as before, I don't need to do anything to my Model class. That's called person, because all the code that's going to be getting the string and outputting it, is already available in the interface. I'll go back to my main class, use static method, and now, instead of putting together this string here, I'll call the interface using the interface's name, PersonInterface, then I'll call the new static method, getPersonInfo and pass in the person object, which is P.

And I'll save the change and run the code. And there's the result. Functionally, it's exactly the same as using a default method, or putting this code into the person class, or having it right here at the top level in the main class. The goal of both default and static methods is just to give you more options in putting together the inheritance model for your application. Using both default and static methods, you can eliminate entire inheritance layers of your application and enormously simplify the coding model, making the application easier to code and to maintain.

# 3. Using Lambda Expressions and Method references

## Traversing collections with streams

Java's collections framework lets you easily manage ordered and unordered collections of data in your applications, using interfaces like list and map, and classes like arraylist and hashmap.The collections framework has continually evolved since its first introduction. And in Java8, we now have a new way of managing, traversing, and aggregating collections with the stream API.A collection based stream isn't like an input or an output stream.

Instead it's a new way with working with data as a whole instead of dealing with each item individually. When you use streams, you don't have to worry about the details of looping or traversing. You create a stream object directly from a collection. And then you can do all sorts of things with it including traversing, filtering, and aggregating its values. I'll start with this example in the project TraversingStreams. In a class code SequentialStream, in Java8 there are two kinds of collection streams known as sequential and parallel streams.

A sequential stream is the simpler of the two and just like an iterator it will let you deal with each item in a collection one at a time. But with less syntax than before. In this code, I've created an array list of people, cast as a list. And it has three instances of a complex object, a class named person. Then I'm using a predicate to declare a condition, and displaying people that only satisfy the condition. From lines 27 to 32 in the display people method, I'm traversing the collection, looping through the data, and testing each item one at a time.

I'll show you how to refactor this code using a stream object. First, I'm going to comment out these lines of code. Now, below the commented code, I'll start with the collection object. People.And then I'll call a new method called stream. A stream object, just like the collection itself, has a generic declaration. If you get a stream from a collection, the items in the stream are of the same type as the collection itself. My collection has instances of the person class so the stream uses the same generic type.

You call stream as a method, and now you have a stream object that you can do things with. I'll start with a simple call to the four each method, and this will require a Lamda expression. I'll pass in the argument. That's the item in the list that I'm dealing with on this time through the iteration. Then the Lambda operator and then the implementation of the method. And I'll use simple system output and I'll output the person's name. I'll save and run the code and there's the result. Because I'm not filtering anymore, I'm displaying all of the people in the list.

Now, once you have a stream this is how easy it'll be to use a predicate object. When I use the for each method and dealt with each item one at a time. I had to explicitly call the test method of the predicate. But using a stream you can call a method named filter. That expects a predicate object, and all predicates have a test method, and so it already knows how to call that method.So, I'll break up this code a bit. I'll move the call to the .for each method down a couple of lines,and then on the empty line in the middle, I'll call the new filter method.

The filter method expects an instance of the predicate interface. And I'll pass my predicate object in. The filter method reterms the stream but now the filtered version, and from there I can call the for each method. I'll run the code and now I am only displaying items from the collection that satisfy the predicate condition. You can do lot more with the streams. Take a look a the documentation for streams in the Java8 API docs.

And you'll see that in addition to filtering you can also aggregate and do all sorts of other things with streams. Before I conclude this demonstration though, I want to show you a very important distinction between sequential and parallel streams. One of the goals of the stream API in Java8 is to let you break up processing on a system that has multiple CPUs. This multi CPU processing is handled automatically by the Java runtime. All you need to do is turn your sequential stream into a parallel stream.

And there are a couple of ways of doing that syntactically. I'll make a copy of my sequential stream class. I'll go to my package explorer, and I'll copy it and paste it. And I'll name the new class, parallel stream. And I'll open the new class. In this version, I'll get rid of the commented code. I don't need that anymore. And now here are two ways of creating a parallel stream. One approach is to call a different method from the collection. Instead of stream I'll call parallelStream. And now I have a stream that will automatically be broken down and allocated to different processors.

I'll run the code and I'll see that it's doing exactly the same thing, filtering and returning the data.Here is the other way to create a parallel stream. I'll call this stream method again. And then from the stream method I'll call a method named parallel and that does exactly the same thing. I start with the sequential stream and I end up with a parallel stream. It's still a stream. It can still filter, it can still process in exactly the same way as before. But now it'll be broken up where possible.

There isn't any clear prescription for when to use a parallel stream over a sequential stream. It depends on the size and complexity of your data, and the capabilities of the hardware. The multi CPU system that you're running on. The only recommendation I can give you is to try it with your application and your data. Set up benchmarks, timing the operation. Use a sequential stream and use a parallel stream and see which works better for you.

**Creating streams from collections and arrays**

Java SE 8's stream API is designed to help you manage collections of data, that is objects that are members of the collection's framework, such as array lists or hash map. But you can also create streams directly from arrays. In this project, creating streams, I have a class named ArrayToStream. And in its main method, I've created an array of three items. And they're each instances of my complex object, the Person class.

This class has setters and getters for the private fields, and the get info method, to return a concatenated string. Now if you wanted to use a stream to process this array, you might think you would need to convert it to an array list, perhaps and then from there create the stream. But it turns out there are a couple of ways to go directly from an array to a stream. Here's the first approach. I'm not going to need these three lines of code that I'm using to process the data. So I'll comment those out. And then down here, I'll declare an object for the type is stream.

Stream is an interface, which is a member of java.util.stream. When I press Ctrl+Space and select it from the list, I'm asked for the generic type of the items, that the stream will manage.And these will be items of type Person, just like the items in the array itself. I'll name my new stream object, stream, in all lower case. And here's the first way to create the stream. Use the stream interface again, and call a method named of. Notice that there are a couple of different versions.

One that takes a single object, and one that takes a series of objects. I'll use the one that takes one argument, and I'll pass in my array, people, and that's all I need to do. Stream.of means take this array and wrap it inside a stream. And now i can use lambda expressions, filters, methodreferences and other things that work on Stream objects. I'll call the stream objects for each method and i will pass in a lambda expression, i'll pass in the current person and then after the lambda operator, i'll output the person's information.

Using the object's getInfo method. And I'll save and run the code and there's the result. I'm outputting the items in the same order in which they were placed in the array. So, that's one approach using Stream.of. There's another approach that does exactly the same thing. I'm going to duplicate that line of code, and comment out one version. And this time in using Stream.of, I'll use a class named Arrays, which is a member of the package java.util.

And from there, I'll call a method named stream. Notice the stream method can be wrapped around arrays of a variety of types. Including both primitives and complex objects. I'll save and run that version and the stream does exactly the same thing as before. So either Stream.of or Arrays.stream will do exactly the same thing. Take an array of primitive values or complex objects and turn them into a stream, that you can then use with lambdas, filters, and method references.

**Aggregating stream values**

I've previously described how to use a stream to iterate over a collection. But you can also use streams to aggregate items in a collection. That is, calculate sums, averages, counts, and so on. When you do this kind of operation, it's important to understand the nature of parallel streams. So I'm going to start this demonstration in the project Aggregating Streams. And I am going to first work with the class Parallel Streams. In this class's main method I've created an array list containing string items.

I'm using a simple for loop I've added 10,000 items to the list. Then on lines 15 and 16, I'm creating a stream and using the for each method and outputting each stream one at a time.When I run this code, I get an expected result. The items are output to the screen in the same order in which they were added to the list. Now let's see what happens when we turn this into a parallel stream. As I described previously, I can do this either by calling the parallel stream method or by taking the results of stream and passing those to parallel.

I'll do the latter. Now I'm working with a parallel stream, that is a stream that can be broken up and the work load split among multiple processors. I'll run the code again and watch what happens, notice that the last item printed is not the last item in the list. That would've been 9,999. And if I scroll around in the output I'll see that the processing is jumping around in someway. What's happening is that the run time is arbitrarily splitting the data into blocks.

And then handing each block to an available processor. It's only after all of the blocks have been processed that my next bit of Java code would be executed. But internally, within the call to the foreach method, all of this work is being split up as needed. Now this might or might not provide a performance benefit. It depends on the size of your data set. And the nature of your hardware. But one of the things that this example shows you is that if you need to process items sequentially, that is one at a time in the same order in which they were added to the collection, then a parallel screen might not be the way to do it.

Sequential streams can guarantee they're working in the same order every time. But a parallel stream, by definition, is going to do things in the most efficient way possible. So parallel streams are especially useful when you're aggregate operations. Where you're taking into account all of the items in a collection and then creating a some sort of aggregate value from them. I'll show you examples of counting items in a collection, averaging them, and summing them using strings.

In this class, count items in the main method, I'm starting with the same basic code. Creating 10,000 strings in a list. And then there's a for each method that's looping through and handling them one at a time. In this example, instead of processing each string individually, I instead want to count them up. So, I'll comment out that code, and here's the code I'll use. Since I don't know exactly how many items are going to be in the collection. I'll cash the result I am about to create as a long integer.

And I'll name it Count, and I'll get it's values by calling strings. That's my collection, .stream, .count, and this returns a long value. Then I'll use system output and I'll report the result. With count: and then I'll append my result. I'll save my changes and run the code and there's the result. The count of the items in the collection is almost instantaneous. Now to make this a little bit more dramatic I'll add a couple of zeros here and now I'm dealing with 1,000,000,000 strings.

I'll run the code again and the result comes back again almost instantly. Now watch what happens if I instead parallelize the string. I'll add dot parallel here,. And then I'll run the code, and it takes a little bit longer. Now, I could benchmark how long it's taking these operations tohappen, by capturing the current time stamp before and after the operation. And then doing a little math. And what it would show might differ from one system to another. But in my experience when dealing with these sorts of simple collections containing simple values there isn't much benefit to parallel streams.

Your mileage may very though. And I encourage you to do your own benchmarking. But that's how you would do a count. Let's take a look at summing and averaging. I'll go to my class sum and average. This time, I have a list of three person objects, each with a different age. And my goal is to get the sum of the three ages, and the average of the three ages. I'll add a new line of code after all the instances of the person class have been added to the list. And I'll create an integer variable that I'll name Sum.

I'll start by getting a stream, using people.stream. From there I'll call a method called Map to Int.Notice that there's a Map Method. Map to Double and Map to Long as well. The purpose of these methods is to take complex oject and extract a simple primitive value from it, and create stream of those values, and you do this with Lambda expression. So, I'll choose Map to Int because the ages of each person are integers.

For the Lambda expression, I'll start with a variable that will represent the current person. Then the Lambda operator and then an expression that returns an integer. I'll use P. get H. This returns something called an int string or a string of integers. There's also a double string class and a few others. Now from this stream because I already knew it was a numeric value I can call a method named sum. And that's it. I've now summed up all the aged values from all the personals object in my collection.

With a single statement, I'll output the result using System Output. My label will be total of ages and I'll append to that my sum. I'll save my code and run it. And the total of all three ages is 151. Averaging these values is very similar. But because whenever you're doing averaging your doing divisions, and you might get a divide by zero problem, and so, when you do an average, you'll get back something called an Optional Variable.

And there are a number of types you can use for this. For my averaging, I'm going to expect a double value to come back. So, I'm going to create a variable, called OptionalDouble. Notice that there's also Optional Int and Optional Log. I'll name my variable Avg, for average. And I'll use this same sort of code I just used to get the sum, starting with people.stream. And then from there, I'll use Map to Int again. And I'll pass in the same lambda expression that i use last time, and then from there I'll call the average method.

Now with an optional double object, before you process it you should always make sure that it actually has a double value and you do this with a method named is present. So, I'll start off with an if else code template. And I'll set my condition to Avg.is present. If that condition is true, I'll use System Output. And I'll label this just Average. And I'll append my average variable. In the else clause I'll simply say that the average wasn't calculated.

Now in this example I know it will be successful because I've provided ages to all three people but that won't always be the case. Like I said if you end up with a divide by zero situation you might not get a double value back. I'll save and run the code, and notice that with the optional double class, it's a complex object. So the type is wrapped around the actual value. I'll go to this code, where I'm referencing the object directly, and I'll call it's get as double method.

And now I'll get back a primitive double value. I'll run the code again and now the result is what I was looking for. So using streams and lambda expressions, you can easily calculate aggregate values from collections with a tiny, tiny amount of code.

# 4. Using Lambda Expressions and Method references

## Calculating time spans with Instant and Duration

JAVA SE8 includes a complete new API for managing date and time values. The classes that actually hold the data in this new API are all immutable and thread safe. So that means you don't have to worry about passing objects around in a multi threading environment. And if you're using them in parallel streams, everything will always work perfectly. All of the classes in this new API are members of the package java.time. And I'll start with two of the core classes named Instant and Duration.

I'm working in a project named InstantAndDuration. And my main class is named main. And I'll place all this code inside the main method. The first class I'll describe is named instant. I'll type the name of the class and press Ctrl+Space, and choose the class from the java.time package, and it's imported above. An instant object represents an instant on the Java timeline. Just as with the date class which is a part of the old way of managing dates and times, an instance represents a number of milliseconds.

Since the Java epoch time, January 1st 1970. To declare an instant object, I'll declare it with its type, and give it a name of start. And then I'll call a static method of the instant class called now.And this represents the moment on the current machine when the code was called. Then I'll output that value as a string, using standard system output. Starting off with the date in year, month, date format and then the time after the letter T.

Once you have a moment in time, you can use it to calculate a difference between that and another moment in time. So I'll create another instant, which I'll call end. And I'll get it's value from the method now as well. Then I'll use system output, and output that value. Notice that there is a slight difference between the two values, and that's the amount of time it is taking on my system. To process this line of code, that's outputting the start value. If I were to move that line of code down, so I wasn't doing any other processing between the two calls to the now method, the two values would be identical.

Or they might be off by a thousandth of a second. Now, I'll show you how to calculate the difference between these two values. When you compare two instants to each other, you'll get an object called a duration. It's represented by the duration class, which is also a member of Java.time. I'll name this object elapsed. An I'll call a static method of the duration class called between. Notice that it's looking for objects typed as something called Temporal. The Instant class is a sub-class of Temporal.

I'll pass in start and end as my two temporal values. And then I'll output the difference. I'll pass in a literal label of elapsed, and then I'll pass in my variable. That duration object starts with the letter p and then t for time. This is again an ISO formatted value. And then it shows me zero seconds. Well let's see what happens if we toss in a call to the sleep method. I'll place the cursor here between the start and end calls. And I'll use the thread class.

I'll press the period, and then press Ctrl+Space. And I'll dismiss my problems dialog. And then I'll call the sleep method and pass in a value of 1,000. Meaning sleep for one second. The sleep method can throw an error, so I'll use a quick fix, and I'll add a throws declaration to the main methods signature. I'll save and run the code, and I see that my lapse time is now 1.001 seconds. You can never really count on things being exact, it all depends on what's going on, on the processing computer.

Next, I'll take this printline call, and move it back to its original location. So now after I get the start value, I'll be executing a printline command. And I'll be sleeping for one second. And I'll run the code. And now my lapse time is 1.057 seconds. To make this a little bit more readable, I'll add a call to the method of the duration object using elapsed.to millis. That means, get the milliseconds equivalent. And I'll append to that, milliseconds, and I'll run the code.

And now I see, a readable value, of 1,054 milliseconds. So, that's the instant class and the duration class. Two of the core classes, of the new date time API, in Java SE8.

## Representing date and time values

I previously described how to use the instant class in the new date time API to represent a moment in the Java timeline. Here are three more useful classes to represent parts of dates and times. They're called local date, local time, and local date time. Lets say, for example, that you only want to represent a date value. And you don't care about times or seconds or milliseconds but only the current date. Create an instance of a class named LocalDate.

I'm working in a project name DatesAndTimes that has an empty class with a main method. I'll start with the name of the class Local Date. And when I press Ctrl + Space, I'll choose the class from the java.time package. I'll name the object currentDate and I'll get its value with localDate.now. Notice that there's consistency in the syntax between working with an instant, a date, a time and a date time.

To get the current value on the current machine, you always use the now method. Now, I'll output that date in its default format. I'll save and run the code, and it shows me the date in year-month-date format. You can also create a date object using specific year, month, and date values. Once again, I'll create an object typed as LocalDate. And I'll name this one specificDate.To get this value, call LocalDate.of.

And there are a couple of available versions. I'm going to use the one that takes three integer values. They aren't named in the documentation but they represent the year, the month, and the day. I'll patch in the values of 2,000, 1, and 1. Now, in the older version of the date time API using the date class. When you were dealing with months, you always had to do it with a 0 based off set. So for January you'd use 0, for February 1 and so on.

And that wasn't particularly intuitive. In the new day time API, everything is 1 based. So 1 means January, 2 means February and so on. Just as you would normally expect. I'll once again use system output, and this time I'll, I'll put the new specific date. And when I save and run that, I get the value that I put in, January 1st, 2000. If you only want to represent a time value, use the LocalTime class, I'll type the name of the class and import it, I'll name the objectcurrentTime and I'll get its value from LocalTime.now.

Again, using the same sort of syntax as with localdate and with instant. Then I'll use system output. And I'll output that current time. The default value for the time is in 24 hour notation, and it shows the hour, the minute, the second, and the milliseconds. I'll also create a specific time value. I'll use LocalTime. I'll name this specificTime. And, just as with the local date class, I'll call the method named of. Again, there are a number of different versions taking varying numbers of arguments.

I'll use the version that's looking for three integer values and I'll type in 14, 0 and 45. And then I'll output that value to the console. And there's the result. 14, 00 and 45 seconds. Notice that because I didn't provide a milliseconds value the formatted version of that time doesn't show values after the dot. Finally, I'll show how to use the LlocalDateTime class.

I'll type the name of the class and import it. I'll name this object currentDT. And I'll get its value from LocalDateTime.now. When you output a date time value, you'll get a long format ISO date time. Starting with the date, and ending with the time. And if there are milliseconds in the value, they'll be displayed. And finally, I'll create a specific date, time and I'll do this by combining my specific date and my specific time.

That code will look like this. I'll create a LocalDateTime object. I'll name it, specificDT and I'll call LocalDateTime.of again. And this time, I'll use this first version of the method that accepts a local date object and a local time object. You can also construct your date time value from combinations of years, months, dates, and time values. I'll pass in my specific date and my specific time. And then I'll output it to the console. And when I run that code, my specific date time is a combination of my specific date and my specific time.

So, those are the three classes that you can use to represent date and time values on the local machine in the current time zone. There are also classes you can use to get time zone sensitive values. And I'll describe those in a later movie.

## Formatting date and time values

I've previously described how to use the LocalDate, LocalTime, and LocalDateTime classes to represent time values. To present this information to a user, you'll need to format it. And for that purpose, there's a new class named DateTimeFormatter. I'll show you how to create formatters using some simple syntax. And then how to do very custom work using a class called Daytime Formatter Builder. I'm working on a project called FormatDateTime, which has an empty main class with a main method.

First, I'll create a date. I'll give it a type of LocalDate, making sure to import that class. And I'll name it currentDate. And I'll get its value from LocalDate.now. Next I'll create a formatter object.I'll type the name of the class, DateTimeFormatter, and select it from the JAVA.time.format package. I'll name this object DF. Now there are a number of ways of creating a formatter. One of the simplest is to use a constant of the DateTmeFormatter class.

I'll once again type in DateTimeFormatter. And then after I type in the period I see a list of all of the available constants. I'll choose ISO Date. And this will provide the default formatting for this object. Then, I'll use System Output. I'll call the formatted object format method and pass in the date object. And here is the result. I'm outputting the value in year month date format. With the month and the date padded out to two characters each. Now you can do the same sort of thing with times and date times.

I'll take this bit of code and I'll duplicate it a couple of times and I'll make some changes. In the second version, I'll change the type from local date to local time. The object name to current time, and the name of the class I'm using to get the value to local time. I'll change the name of the date time formatter from DF to TF for time formatter. And I'll change the constant I'm using to ISO Time. And then I'll change the object that I'm formatting. I'll be sure to import the LocalTime class.

And then I'll make similar changes to the third version. The class that I'll be working with this time is LocalDateTime. I'll be sure to import it. I'll name this object, current DT. And I'll change the class that I'm calling the now method from. I'll change the formatter to DTF for DateTimeFormatter. And I'll change the constant to ISO Date Time. And then I'll format the current DT object. I'll be sure I'm using the right formatters in each version of the code. I'll save the change, and run the code.

And there are the three formatted values. Now so far, I haven't really accomplished that much,because I've used the constants that represent the default formatting. But let's take a look at some custom formats that are available. I move down to below the existing code. And i'll create another DateTimeFormatter, i'll name this one f\_long for the long date format and i'll get its value by calling a method of the DateTimeFormatter class called a Localised Date.

Notice that there are methods for date, time and date-time, with a variety of arguments. I'll choose this one, of localized date, and I'll pass in a constant of a class called,FormatStyle. Be sure to import this class,. And then after you type the period, you'll see that there are four constants available. Full, long, medium and short. I'll choose the long version and then I'll output the formatted date by calling F \_ long.format and I'll pass in the current DT object.

When i run this code i get the long version of the date. I'll show you another version of this byduplicating these two lines of code and for this version i'll change the formatter name to f\_short,i'll change the constant I'm using to short also. And I'll change the name of the formatter that I'm calling. So the long version is the name of the months spelled out. A comma after the date, and then the year in four digit format, and the short version at least for the current locale, is the month and date, without padding, with slashes separating values, and a two character year.

And next, I'll show you how to use locales. I'll create a couple of strings. The first will be called fr\_ short, for French, in short format. To get that value i'll call my f\_short formatter and then i'll will call method name withLocal. To get a locale value i'll use the Local class, this is an existing class that's been available in previous versions of Java. It's a member of the package Java.util.

And then, I can call one of the many constants representing various locales. I'll use French. And then from there, I'll call the format method, and pass in the current date time. I'll duplicate that line of code and for this version I'll use fr\_long. I'll use the long formatter and otherwise the code will be the same. And then I will output those two values. Fr \_short and fr \_long. And here's the result. Notice for fr\_short that the month and the day are reversed from the US version.

And that's because in Europe, the date is stated first, and then the month, and then the year.And when I use the long version I get the months spelled in French. Finally I'll show you how to build completely custom formatters using a class called the date time formatter builder. It uses the builder design pattern, where you can call multiple methods, each returning an instance of the current builder. I'll type the name of the class and make sure that it's been imported. And I'll name the object B. I'll instantiate it with the new key word, and the constructor method.

Now, at the end of that code I won't put in the semicolon because I want to immediately call a series of methods that let me build the formatter from scratch. I'll start with a method named Append Value. Notice there's Append Instant, Append Literal, Append Localized and many many others. I'm going to call a method named append value that accepts an instance of a class named Temporal Field and then I'll use an enumerator named Chrono Field. Which is extended from that Temporal Field.

And from there I'll use a constant name month of the year. Next I will append a literal value.This can be any character or any string. And just to make this thoroughly unique, I'll put in a couple of pipe characters. Now I'll take these two lines of code and duplicate them for the second version instead of month of year. I'll put in day of month. Notice there's also day of week and day of year. Then I'll duplicate that line of code and move it down. And I'll finish the expression with ChronoField.year.

Once you've created the builder object, you can get the formatter. I'll create new object, typed as DateTimeFormatter. I'll name it F for Formatter. And called builder objects, to formatter methods, and then finally I'll format the current date time value. I'll use system output and call F.format, and pass in current DT. And now when I run my code, I get completely customized format.

You can use the DateTimeFormatter builder to build any format you like. And because it uses the builder design pattern, it's easy to create and to maintain the code.

**Supporting time-zone offsets**

The new date time API offers a number of classes that let you manage time zones. Creating day time objects that are offset from Greenwich Mean Time, by certain number of hours, or by particular locations, and calculating differences between time zones. I'm working in the project named TimeZones. And in its main class, I've created a DateTimeFormatter and a LocalDateTime object. The LocalDateTime represents the current date and time on my system, in my time zone.

And that's Pacific Time because I'm on the West Coast. And then I'm outputting a formatted value to the console. I'm outputting the value using a short format. And in USA notation, it's month, day and year. In order to represent a time zone based date time value, use the class ZonedDateTime. Just like LocalDateTime, it's immutable and thread safe. I'll type the name of the class, and then press Control Space to add the import statement. And I'll name the object gmt for Greenwich Mean Time.

There are a few different ways of creating this object. I'll show you how to create the object calculating an offset from Greenwich Mean Time. I'll use the ZonedDateTime class again, and after I type the period, I'll see that there are many methods available. I can call now again, to get the date time value in my area. I can call of methods that let me do various calculations. I can parse strings, but I'm going to use this version of the now method. I'll pass in an instance of the zone Id class.

A zone Id represents a certain number of hours offset from Greenwich Mean Time. And I'll get that value by calling a method named ZoneId.of. And I'll pass in a literal string of GMT plus zero. That means, show me the current date and time value in Greenwich Mean Time. Now I'll duplicate my code that's outputting the value to the console. I'll move that down, and I'll change this version to output gmt. I'll run the code, and there's the result.

I'm on the West Coast of the United States in Pacific time, and right now, Greenwich Mean Time is seven hours ahead. Here is another approach to getting a ZonedDateTime. Lets say you wanted to get the ZoneDateTime in New York. There are many built-in strings, or constants, that will let you name particular locations, and you'll get back the correct ZoneId for that location, and you won't have to worry about the math yourself. I'll create another ZonedDateTime object, and this time i'll name it ny for New York, and I'll get it value by callingZonedDateTime.now, and again I'll pass in ZoneId.of, but this time i'll pass in a string of America/New\_York.

Make sure to spell this string exactly as you see it here. I'll create a line of code to output that value. I'll save the change, and run it. And New York is on East Coast time, three hours ahead of Pacific time. To find out about all of the available strings, you can call a method of the ZoneId class called get available ZoneId's. You'll get back a set. I'll type Set and press Control Space, and then choose set from Java.util.

And I'll set the generic type of the items in this set to String. I'm name the set zones. And then I'll call the method, ZoneId.getAvailableZoneIds. Then I'll loop through the strings with the forEach method. If you see that error, just dismiss it, and then I'll pass in a Lambda expression.So I can deal with each of the items in turn. When I run that code, I see all of the available strings. Now, there are so many it might be hard to find the one that you're looking for. So let's say that I wanted to look for London.

And use the time zone for that particular location, at this particular time of year. As I showed earlier in the course, I could use a predicate to search the strings. I'll create a predicate object.And I'll set the generic type to string. And I'll name the object condition. Then I'll implement the predicate with a lambda expression. I'll pass in str, and then I'll implement the predicate with a condition. STR.contains, and I'll pass in a string of London.

Then I'll refactor my Lamba expression. I'm going to wrap System.out.printline in braces. Then I'll expand the code to make it a little bit easier to work with. I'll add the semi colon at the end ofthe print line, and then I'll create an if statement. And I'll set the condition to, condition.test, andI'll pass in z for the current zone. I'll move the printline statement, to within the condition, andnow i'll only printout strings that match my predicate test.

I'll save the change and run the code, and there's the result. I find that the correct string for London is Europe/London. So that's a little bit about working with time zones. Again, use the ZonedDateTime class instead of LocalDateTime to represent values that you can modify and calculate against. The ZoneId represents an off set from Greenwich Mean Time. And there's also a class called Zone Offset that you can use to calculate different time zones against each other.