Contents

[**Course Overview** 3](#_Toc78814635)

[**Course Overview** 3](#_Toc78814636)

[**What Are Collections and Why Use Them?** 3](#_Toc78814637)

[**Introduction** 3](#_Toc78814638)

[**The Array Problem (Live Coding)** 3](#_Toc78814639)

[**Course Outline** 5](#_Toc78814640)

[**Collection of Collections** 5](#_Toc78814641)

[**Collection Behaviors** 6](#_Toc78814642)

[**Collection Concepts (Live Coding)** 7](#_Toc78814643)

[**Collections with Iteration Order: Lists** 8](#_Toc78814644)

[**Introduction to Lists** 8](#_Toc78814645)

[**Key Features** 9](#_Toc78814646)

[**Shipments Example (Live Coding)** 9](#_Toc78814647)

[**Shipments Example 2 (Live Coding)** 11](#_Toc78814648)

[**Shipments Example 3 (Live Coding)** 12](#_Toc78814649)

[**Implementations** 13](#_Toc78814650)

[**Conclusions** 15](#_Toc78814651)

[**Collections of Pairs: Maps** 16](#_Toc78814652)

[**Introduction** 16](#_Toc78814653)

[**Why Use a Map? (Live Coding)** 16](#_Toc78814654)

[**Map API** 17](#_Toc78814655)

[**Views over Maps (Live Coding)** 18](#_Toc78814656)

[**Java 8 Enhancements** 19](#_Toc78814657)

[**Java 8 Demo (Live Coding)** 20](#_Toc78814658)

[**Implementations** 22](#_Toc78814659)

[**Correctly Using HashMap** 23](#_Toc78814660)

[**Java Streams** 24](#_Toc78814661)

[**Introduction** 24](#_Toc78814662)

[**Live Coding Streams** 24](#_Toc78814663)

[**Operations on Streams** 26](#_Toc78814664)

[**Enter the Collector (Live Coding)** 27](#_Toc78814665)

[**Conclusion** 28](#_Toc78814666)

[**Collection Operations and Factories** 30](#_Toc78814667)

[**Introduction** 30](#_Toc78814668)

[**Factory Methods (Live Coding)** 30](#_Toc78814669)

[**Factory Method Options (Live Coding)** 31](#_Toc78814670)

[**Factory Methods** 32](#_Toc78814671)

[**Collection Operations** 34](#_Toc78814672)

[**Collection with Uniqueness: Sets** 35](#_Toc78814673)

[**Introduction** 35](#_Toc78814674)

[**Set Features (Live Coding)** 35](#_Toc78814675)

[**Hashcode and Equals** 36](#_Toc78814676)

[**Set Implementations (Live Coding)** 37](#_Toc78814677)

[**Conclusion** 39](#_Toc78814678)

## **Course Overview**

### **Course Overview**

Hi, everyone. My name is Richard Warburton, and welcome to my course, Java Fundamentals: Collections. I'm a director at Opsian, a best‑selling book author, and also recognized as a Java Champion. As any Java developer knows, when your coding Java, you use the collections framework all the time and throughout your application. As a result, learning this core skill can bring huge improvements to your Java coding capabilities. In this course, we're going to cover why you should use collections, which collection you should pick in what situation, the APIs and behaviors of lists, maps, and sets, important collection algorithms and how to process data with the Streams API, how different collection implementations work and why you should choose one or another. By the end of this course, you'll know how to use Java collections to process data effectively, and before beginning the course, you should be familiar with basic Java programming constructs. From here, you should feel comfortable diving into Java with courses on Java Fundamentals: Generics and Java Interfaces and Abstraction. I hope you'll join me on this journey to learn Java collections with the Java Fundamentals: Collections course, at Pluralsight.

## **What Are Collections and Why Use Them?**

### **Introduction**

Hello, my name is Richard, and I'm here to teach this course on the Java collections framework. The Java collections framework is a fundamental and essential topic in becoming a strong Java developer. That's why I'm so excited to be teaching this course. No matter if you're a beginner or already an experienced Java developer, you can always learn something new about collections. But first, what are collections? The first thing that comes to my mind when I think about collections is this bag of Roman coins, this collection of old, ancient, Roman coins. You can see all the faces on there. Some of them are happy. Some of them are sad. Some of them are just plain ugly, if we're being honest. But if we think about what our ancient Roman would do if they have this set of coins, they would take them, and they'd put them into a little pouch or a little bag, and that pouch or bag is the collection equivalent in this Roman coin analogy. A collection is like a little container that can have zero, one, or many elements in. That's fantastic, you say. But hang on a minute. In Java, we already have a thing that has zero, one, or many elements in. It's called an array, and it's built into the language. So next up, we are going to talk about the problems with arrays, and we are going to also show you how to introduce and set up the project that you'll be working with throughout this course

### **The Array Problem (Live Coding)**

In this section, we're going to demonstrate the project structure that this module has and also the problem with arrays that motivates us to use collections to begin with. The first thing you should do once you've downloaded the materials.zip file is extract it into a directory that you find convenient on your local machine. Now, this project uses the Gradle build tool, information of which can be found online on Gradle's website This is not a general intro to Gradle module. But if you just want to run the build on the command line in order to confirm everything compiles and builds correctly, you should browse into that directory and run gradlew and then hit Enter, and that will run the build. The first time it may have to download the Gradle tool, and as a result, your build may take slightly longer than you've just seen it take then on my machine. In order to use this project, we're going to be using the IntelliJ IDEA integrated development environment. This is the one I use on a normal day‑to‑day basis when programming. You don't have to use IntelliJ IDEA. You can use Eclipse or NetBeans, if you prefer. I think that IntelliJ IDEA is a very good IDE. I strongly recommend people using it. For this project, we are using Java 11. That's our baseline JVM version, so make sure you have JDK 11 installed. A because we're using JDK 11, we're also using Gradle 7 and that means that we should also be using IntelliJ IDEA 2019.3 or later. So if you're using an old version of IntelliJ, please try and make sure that you've upgrade it to the latest version because you can get some compatibility problems with Gradle 7 otherwise. We import our Gradle project by clicking the Import button on the main menu, browsing to the directory where our materials directory is that we've unzipped, and then selecting the build.gradle file. You'll notice now our project has come up that we've got a couple of directories of code in, src/main/java and also src/test/java for modules where we run JUnit tests. You might need to have a look in each of those directories to find the code. For each of the modules within this course, there is a specific directory 2, 3, 4, etc., that has the number in the name of the package. And you also see if we expand that, we have before and after versions of the code so you can have a look at what the code looked like beforehand and after. The first thing we're going to do with arrays is create an array out of different products here. So our product object is just a regular domain object. It has a name, it has a weight, it has getters for those name and weight values, and it has a human readable toString as well. So if we create an array of Product, which we're going to call products, there is a nice, convenient syntax that comes with Java for creating that array. So we're going to create an array with our door and our floor panel object in. And the first thing I'm going to do here, because creation actually works quite well with arrays, is try and print out that products array. So if I run that, we can see the output here from our program is just this completely unreadable [L, Product, blah, blah, blah, blah nonsense. And that's the first problem we have with arrays here. They are not very easy or convenient to code against. They can be difficult to debug because they don't have a human readable toString implementation. So if we want to actually have a human readable toString, we are going to have the pass the toString, Arrays.toString method there in order to do that, and now we can see we've got our product with a wooden door and a floor paneling. Let's try and do another common task with our arrays, which is to add an element into the array. So the first thing you might think is arrays are indexable. I can add a new product in here. We'll call it a window. So I'm going to take our window product and add it in at element 2 And if we rerun that code, we can see we get a big exception problem here, Index 2 out of bounds for length 2. Okay, that is no good at all. Arrays themselves are not resizable in Java. They are a fixed‑length array like in C, and that means as a result, if you ever want to add an element to an array, you need to create a new array, copy the element over, and add the new element in place. Now, I have conveniently created an add method here, and our add method, we can see if we run this code, it works okay, so no exception this time round. And our first time it printed out there was no window, and the second time around, we had our glass window printed in there. So if we have a look at the way our add method works, we have to take this copy of the array, set its new length to be bigger, assign the value of the product into the new length, add its new index, and return that array. Very inconvenient, not built in, very hard to use. Collections, on the other hand, have nice convenient add methods, they support lots of different things, and obviously they're a lot more flexible than arrays as well. So, for example, with arrays, there's no way of enforcing business domain level constraints within an array, like our Duplicate operation. You want to ban duplicates from collections? We could just use a set collection that you'll see in one of later modules in this course. With arrays, if I try and add in another window into our products array and then if I print it out, and if I rerun this method, we can see that we get a duplicate here. We have two glass windows in our output, so that is no good at all, and that's very hard to implement arrays. So arrays are a low‑level programming construct, they're a reasonable thing to have in the core language, but they're not flexible enough for our use cases. And if we were to use arrays directly in our business code, we would spend a lot of time working on adding custom functionality like adding or fixing the duplicates case that we can just get out of the box with collections. Much better.

### **Course Outline**

This section will outline the course contents and tell you what you're going to learn and understand throughout the course of this course. Firstly, implementing data structures is really hard. That's why a lot of the big tech companies love to provide data structure‑based interview questions. There are lots of corner cases, it's very hard to get everything right, and it's very hard to get them performing well. That's why it's fantastic that the collections framework ships with the JDK itself. Trying to reinvent the wheel when you have a built‑in collections framework that does that work for you might be a good fun task for a developer, but it's not a good use of your time, and you're likely to write buggy code, compared with the well‑tested code that ships with the JDK. Now the data structures that ship with the Java collections framework have very diverse properties, and that's good because different use cases require different diverse properties. So some data structures define ordering within the collections so you know which element within the collection comes after another. Some data structures, maps specifically, provide pairs of elements, keys, and values that are associated together. And finally, some data structures provide uniqueness, so you know that each element is only within the data structure once. This course has a series of different modules. This is the first module. It's just introducing you to the concept of collections and how you can program against the API. Next, we'll look at the module on lists. These are collections with iteration order. They're the most commonly used collection, and you've probably hit them already in your programming environment. Then we shall look at maps. Maps are collections of pairs. Then we'll look at Java streams, which is an API that was introduced in Java 8 that makes it easier to program and perform operations using the Java collections framework in a more functional and modern style. There we'll look at different operations on collections and certain factory methods for creating collections more conveniently. And finally we'll look at sets. Sets are another type for collection, less commonly used than lists and maps but still common enough to warrant consideration in this introductory course.

### **Collection of Collections**

As you may have noticed, I keep on referring to the Java collections framework, and that's because there's more than one collection that ships with the JDK. So let's investigate this collection of collections and see what the most appropriate collection is to use in a given situation. The collections framework's top interface is the collection interface, and that's the interface which most of the collections extend from. The most commonly used collection is the list collection, so that is a collection that has ordering of elements, and each element has a given index. So the first one is 0, the second one, 1 etc. Then there is the set and SortedSet and NavigableSet interfaces that we see in the collections framework. Set has uniqueness as a key property. Then there's the queue and double‑ended queue, or deque, interfaces. We won't really be covering these in detail in this collections course, but they're a interesting collection to have a look at. They have properties around the way in which elements are added and removed from the collection, so you can do things like implement first in, last out or even actually first in, first out behaviors using these collections. Then there is the map and its SortedMap extension. Map is a collection of pairs. Now, each collection has two or more than two, in many cases, different components. Firstly, there are the interfaces, and then there's the implementation of those interfaces, So the interfaces here, like list and set that we've seen on the previous slide, they define multiple different data structures that can implement this interface. They define the functional characteristics of the collection, so they define things like ordering, uniqueness. They don't define how the collection is implemented under the hood. If you're declaring a variable as a collection in Java, you always want to prefer the interface as the type on that variable where possible. Often there is a particularly popular implementation of that interface. Implementations define specific data structure, a specific way of implementing a given interface. Each data structure has a different set of performance characteristics, so it's important you know what the right data structure is to choose when you want a certain interfaces functional characteristics as well. Implementations are concrete and instantiable, so whilst you'll declare a variable type as list, you would always instantiate it with a specific implementation. Let's have a look at the different implementations that we have for the collections framework. For our list interface, the most popular implementations are ArrayList and the LinkedList. For set we have a HashSet, and its SortedSet compatriot has a TreeSet implementation. Queues and Deques have PriorityQueue, LinkedList, and ArrayDeque implementations. And then over to the side, map, which, as mentioned, doesn't actually extend the collection interface, though it is part of the collections framework, has the HashMap implementation, and SortedMap has the TreeMap implementation. So when do we actually choose how to use these collections? Firstly, you want to ask yourself the question, are your elements keyed or they just elements on their own? And if the answer to that is yes, you want to ask yourself, is the order in which you iterated over those elements important? And if not, just use a map. Otherwise, use a SortedMap. Now if we go back up to our original question about whether things are keyed, if you answered no to that question, then you want to ask yourself, did the elements have to be unique within that collection, or can you have duplicates within it? If the answer is yes, then you want to ask yourself, is order important? And if the answer to that is no, you can just use set. And if order is important, you want to use the SortedSet interface. Finally, we want to ask ourselves, does insertion order matter? And if insertion order matters and our collections don't have to be unique, we want to use a deque, or double‑ended queue. Otherwise, you want to use the list interface. That's a very, very simple guideline that will provide you with a very, very useful set of heuristics as to which interface to pick in your particular project. Next, we're going to look at the API of the collection interface and the common components that all collections have and how you can code against it.

### **Collection Behaviors**

Now let's take a look at the common operations of behaviors that different Java collections have in common. Firstly, it's important to note that the collection interface actually extends another interface. That's to say the iterable interface. And the iterable interface allows us to create this object called an iterator. That is the way that we loop over the elements within a Java collection. As we'll see in module five, there's actually new concept in Java 8 called streams that is proving to be a very popular way of performing some of same operations, and often a better way of doing it, but iterators are still covered because they're absolutely key to collections, and you'll see them all over the place in any code that exists in the JDK or in your own code before Java 8. Now, the first set of operations that collections have in common are these, so that includes the size, which is a method that tells you the number of elements in the collection, the isEmpty method, which is kind of a shortcut for the common case of checking whether the size is 0. It also can be a useful method because on certain collections, it can be faster to check the isEmpty method rather than calling the size method. Add ensures that an element is in the collection. So it adds it if it's not there. AddAll adds all the elements of its argument collection to this collection. So that's a way of taking a whole collection of elements and a shortcut for adding them all into your collection. Then there's remove, which removes an element from a collection. RemoveAll removes all the elements of an argument that's a collection from the collection that it's called on. RetainAll that removes any elements that aren't in the collection as an argument. Contains, which tells you whether an element is a member of a collection or not. So true if it's in the collection, false otherwise. ContainsAll does something similar, but with a whole collection of values. So, are all the elements of the argument in this collection, is it a subset? And clear removes all the elements from this collection

### **Collection Concepts (Live Coding)**

So let's take a look here at the different concepts and operations that collections have in common, these common behaviors, and see how we can apply them in a live coding session. Firstly, we'll create a collection of products. That <Product> construct here means that the only values that are contained within this collection are going to be types of products or subclasses of the Product class, and we'll initialize it with a new ArrayList. Now, what this is actually called here is a generic type parameter. A generic type parameter is something that can be explored in more depth in my Java Fundamentals Generics course that is also available on Pluralsight that explains a lot more about what generic type parameters are, everything about generics, everything how you use them are in that core important Java concept, but for now you can just remember that all this means is the only type of object inside of our collection is a Product. So, firstly, we said that one of our problems with arrays was we couldn't print them out, so we're going to print out our Products collection here. Before we do that, we are going to add in a few products, our door, our floorPanel, and our window. Now recall that when we did the same thing with our array, we didn't have a built‑in add operation, and we didn't have a built‑in toString operation, so we couldn't have an easy way of printing things, we didn't have a way of adding things. That is very, very much solved now with this Java Collections API. But that has printed out all of our products on one line, let's suppose you want to have a way of printing them out on different lines, so using a Java loop to look through all the elements in the collection one after another. So the easiest way of doing that is we can use a foreach loop, going over all of our products, and we can print out each product on a separate line. So this code here should be read as for each product in the products collection, and then you specify a type here as well, and I'll just get this to run. Now we can see that we've printed the three products, the Wooden Door, the Floor Panel, and the Glass Window, and they're actually printed out here in the order that we've added them in. That is an aspect of the list collection, as we'll see in the next module, but not every collection. Now something we might want to do here is we might want to take our product, and if its weight is over 20 we will remove it from our products ArrayList. Okay, so we're going to say if the product's weight is over 20, take the product and remove the product object. And if we run this code now, we'll see we get a ConcurrentModificationException here. That ConcurrentModificationException is telling us that you've been looping over this products ArrayList, and you've tried to modify it whilst you were looping over the products ArrayList. So we cannot remove products, we cannot add them either. just don't modify that products collection whilst your iterating over it, at least not using the foreach method. There is a different way that we can loop over our products collection that allows us to remove things, and that is using the iterator. Remember that I said that each collection extends the iterable interface and has an iterator associated with it. Well, we have this iterator concept here, and an iterator is basically like a cursor that lets you go through one by one the elements in your collection, and that is the more traditional way of looping over things rather than this foreach loop. In fact, under the hood, javac converts our foreach loop into our iterator. So the iterator format is to have a while loop, and every time you call iterator.hasnext to say, look, is there another element? And when there isn't another element, your loop will terminate, and then in order to extract that element you call the next method, and that will create our product. If its weight is over 20, we can then say iterator.remove, and at the end we can print out our products collection and see what values it has. And we can see it just has the Glass Window, which makes sense, because that is the only product that has a weight of 20 or less. So you've seen the foreach loop, we've seen our iterator‑based loop here, and we've seen how our iterator‑based loop can do something similar, i.e., iterate over and you can see all those products in turn. Normally, if you're going to iterate over things, I would use the foreach‑based loop, but if you want to remove elements one by one based upon that iterator value, then calling the iterator.remove method is a good way to go about that, and provides a legitimate use case for the iterator concept in Java. So now we have seen the iterator method for removing elements, let's take a look at some of the other operations that we have on our collections interface. Firstly, we have our size method, which tells us the number of elements. And then we also have our isEmpty method. And I'm also going to tell us to quickly check the contains method as well. I'm going to try that on the window. If I rerun this code, we can see we get three false and true. So the size is 3, isEmpty is false, and contains returns true because window is a member of the products collection. So let's also have a look at the remove operation quickly. I'm going to remove the window, and then I'm going to return the contains method again, having removed the window. And so we see the first true is the contains method being true for the window being an element. Then remove returns true, so that's correct, that's saying window has been removed successfully, and it was an element before the removal happened. Then the third value there, the false, is the fact that contains returns false because window is no longer an element. Now we can also create another Collection of Product here. I'm going to call the toRemove collection. I'm going to add the door and I'm going to add the floorPanel to this collection. And then I'm going to take the original products collection, and I'm going to call removeAll operation on toRemove. Then I'm going to print out products. And that will result in us only having the Glass Window here. So removeAll is an interesting operation. Just like addAll and retainAll, it takes another collection as its argument and applies this as a kind of a bulk update operation here, so not just removing one element, but removing all the elements that are in this other collection. And those removeAll, addAll, containsAll, and retainAll operations are very useful when dealing with collections and those kind of bulk operations between different collections. So, a quick summary of what we've learned in this module. Java collections are useful, they're versatile, they're shipped with JDK. There's a large number of different collections that you need to get used to using in the JDK. There are different purposes for all of them, and we've looked at when you want to use specific collections and given you a quick flow chart that lets you see when you might want to use one collection or another. And finally, we've covered common collection features that all the different collection objects have in common. Our next module, hopefully I'll see you there, is covering the list collection, which is the most commonly used collection of them all.

## **Collections with Iteration Order: Lists**

### **Introduction to Lists**

Hello, and welcome to this module on lists, taught by me, Richard Warburton. Lists are the most popular form of collection. They have a defined iteration order, and each element has an index, which is to say a number that defines that iteration order. In this module, we're going to talk about three key areas. Firstly, we're going to look at the key features of lists, what separates and differentiates them from other types of collections, like sets, queues, or maps. Then, I'm going to go through a worked live coding example covering shipments. So a shipment is a collection of products, and we're going to be processing shipments and adding and removing things from them. Then, we're going to look at the different implementations of lists. So we have an array list. We have a linked list. They have different data structures that lie under the hood, and they have different performance characteristics, so we're going to look at when you want to use an array list and when you want to use a linked list.

### **Key Features**

So let's look at the key features of lists. What makes a list a list? Well, lists are really collections that have an iteration order, or at least that's how I'm going to define them in this course. What that means is if you iterate over the elements in a list, don't modify it, and iterate again, you'll see the elements in the same order that you saw them the first way around. Another way of putting that is that every element in the list has an index associated with it, which is an int value that determines its position in a list. So the first element in the list has position zero, the second, one, etc. And it's these indexes that also define the way the API for lists works. So, for example, we initially have the add method on a collection that took only the element and determined no position. Lists, as an extension of the collection interface, inherit that add method, and if you call it, that add method puts the element at the end of the list. But they also have an add method that takes an index, puts the elements into that index, and shuffles anything that may be after that point in the list up by one. There's a get method that lets you read an element out at a given index, so that is the equivalent to, say, an array with the square bracket operator, allowing you to read elements out of an array. Then there is the remove method, which, again, takes an index as a parameter and removes an element at an index. This shouldn't be confused with the remove operation on collection that takes an element and removes that element from the collection. There's also set, which simply updates the element at a specific index. So that is the equivalent of an array's square bracket where you're writing an element into an array. Finally, there's also our bulk update operation, the addAll that takes an index. So, take this entire other collection of objects, put it into our list, and starting at the specified index.

### **Shipments Example (Live Coding)**

The shipments example involves us taking a bunch of different products, and we're going to put those products onto a class called a Shipment. Now, we have in our company two different vans, a light van and a heavy van. The light van is cheaper to run. So if products can be shipped on the light van, we'll put them on the light van. If they're over a certain weight, 20 kg, we'll put them on the heavy van, and our Shipment class is going to take a bunch of products and prepare those shipments into lighter van and heavier van products. We're also going to be able to replace products within the shipment if you want to modify it before it gets sent off. So let's take a look at our code. Firstly, we're going to write this code driven by tests, so we've got a ShipmentTest class that initially all of our tests fail. They all return red. And in this exercise, we want to turn those tests green. Now, if you're following along at home, you might need to change over your Shipment class to point to the version of the Shipment class in the before package, which will look like this, which is what we're going to start coding with, which is to say it compiles against the test. We've got the methods here in the API, but we don't have any of the implementation that goes along with it. Within our ShipmentTest, we can see a few different things. Firstly, we've got our test fixtures of the top, a door, a floorPanel, and a window, so those are our example objects that we're going to be using to code this test. We can also see within our test classes that we've got this assertThat method and a contains method, and our assertThat method and our contains method are part of the JUnit and Hamcrest libraries, so they make assertions of about the code that we're going to run here. If you're not familiar with JUnit or Hamcrest, please take a look at my Introduction to Testing in Java course, also on Pluralsight, which will give you a good explanation of what is needed there, and why you want to write tests, and things like that. So what we've got here, beginning with our test, is we add the door and the window products into our shipment, and then we assert that the shipment contains a door and a window. So that contains method is this statically imported Hamcrest Matcher, and that relies on an Iterable object, so that makes an assertion about something that's it iterable. Let's take a look at our Shipment class. Our Shipment class actually implements Iterable of Product. Now, you can choose to implement a shipment in a variety of different ways. You could, for example, have a shipment that has a list of all of its products and let's you return that list. We've decided here to encapsulate the internal data structures within the Shipment and just make our shipment be an example business domain class, so it's actually going to just implement the Iterable of Product, and that means it also has an Iterator method, which is the only thing in order to implement your Iterable of Product. So let's start implementing code for that first test. We need to have our \_\_\_\_\_ List of Product, and we're going to instantiate that with an ArrayList. Again, remember, these angular brackets and the Product here mean it's a list the only contains Product here, and ArrayList is our implementation. We'll talk about the different implementations of the List interface in the next module. Adding a product is really nice and simple. We add our product. We just delegate to the add method on the backing list, and same with Iterator. We just delegate the backing Iterator method on that list. So we are just really encapsulating the code here. And if we run our first test, we can see that that code is true. So, actually, one of the things we've got here with this Shipment class, this idea of not trying to expose raw collections all over your business domain class, is actually a really good idea because it means that you can change the underlying implementation of those classes, so we could replace a List here with a HashMap or set later on in the implementation of a Shipment class, if we so choose to do, without having our public API break and other classes needing to be modified at the same time, though always, if you can, try and wrap up your collections and keep them private, and keep them within your business domain objects, and use business domain classes to represent business domain concepts. Don't use a List of collection for a shipment. Use an actual Shipment class. That's good Java style. So let's try our shouldReplaceItems test. That obviously fails because even though it implemented the add method and the Iterator, we haven't actually implemented the replace method. Our replace method is going to be an in order replacement. So what we're going to do is we're going to take our products list, and we want to actually find where the old product is in our list, get the index, and then operate on the index. Remember, lists are index‑based collections, so we can find the indexOf our oldProduct, and we'll call that the position, and then we want to take our products class and call set with that position, so that replaces that oldProduct with a newProduct. So let's rerun this test, and we see it now passes. One of the funky things that we have with this code in Java 8 or later is actually there is a replaceAll method that we could have used, so this applies are a UnaryOperator function in order to change all the elements within the list. Now, it's a further exercise at home if you fancy playing around with it to try and implement this same operation using that replaceAll with a function method.

### **Shipments Example 2 (Live Coding)**

And now we're going to continue the shipments example from the previous section. Our next method is shouldNotReplaceMissingItems, so we're going to say replace a door with a floorPanel here, but, actually, we've only got a window in our shipment. So what happens with the replace method when you try and replace an element that's not there? Something we glossed over in the previous discussion, the replace method, is actually that it returns a value, a Boolean value, so it could be true or false. And, in fact, here our test has an assertFalse. So, that means that our replace, if there's a missing element, should also return false. So we've implemented replace here, what happens when we try and run this code? Well, we actually get an IndexOutOfBoundsException: Index ‑1 for length 1. So, why is that? Well, if our old product is missing from the collection, then the indexOf method will return a position of ‑1. So, let's try and check that case here. So firstly, if position is equal to ‑1, what we're going to do here is return false, that is the missing collection case, and, otherwise, we are going to set that position in place and return true. And know obviously ‑1 here, it's a bit of a magic number if we leave it lying in our code. We don't want to have magic constant numbers, like ‑1, lying around when they have a meaning that's relevant to our business domain. So let's create a constant here called MISSING\_PRODUCT, and I'm going to also change that access modifier to be private because it's really part of the internal implementation of the Shipment class and that shouldn't be exposed in the public API. Let's check that our shouldNotReplaceMissingItems test now passes. Fantastic! And I'm also going to rerun the previous test on the replace method just to check that that passes as well. Fantastic! We are cooking on gas here. So let's take a look at our fourth and final test here, and that is our shouldIdentifyVanRequirements test. In order to complete this test, we have to implement the prepare method, and the prepare method builds up the lightVanProducts list and the heavyVanProducts list that we then assert the contents of using the matches at the end of the test. So we're going to put the door, the window, and the floorPanel elements into the shipment, and then we want the window, which is light, to be shipped with the lightVan and the floorPanel and the door to be shipped the heavyVanProduct. Now, whoever's ordered this big shipment of products, they are clearly doing a pretty big renovation or remodel of their house. They've got a door, they've got a window, and a floor panel. Wow, loads of stuff. The question is, do we have to do a big renovation of our code in this class, in our Shipments class, in order to meet these requirements? And the answer is no. But, we do need to make some changes. So there's a few different ways that you could implement the prepare class. So you could iterate over your existing list of products and then kind of just add them if they're below or greater than the values you want, but what I'm going to do here is take our list of products and I'm going to do a different algorithm. I'm going to sort product list, then I'm going to bind the split point, so that's the point at which we need to move things into the heavyVanProducts and not the lightVanProducts. Then, I'm going to create two subviews of the list for the light and the heavyVanProduct. So, this is really using the List API extensively in Java in order to implement this. So, firstly, sorting the product list. Well, if you were using Java before Java 8, there's a method on collections called Collections.sort that takes a products list. And, in fact, you need to provide a way of the sort method understanding how to sort the products, a sort order. So sort orders in Java are defined by this comparator class. We've actually got one built in to our product, which is sorting the product by weight. I'll look at that in just a sec. But that's a bit of a legacy method in terms of implementing it. In fact, IntelliJ even recommends Collections.sort could be replaced by List.sort. So, what does that look like? Well, this is the more modern way of sorting lists in Java that uses a method on the list interface that was added in Java 8 and should be used from Java 8 or later, products.sort by weight. So, what is this sort method taking? It's taking a comparator as an argument. So a comparator is a functional interface, it's an interface with a single abstract method that needs to be implemented. That method is called compare on the comparator, so it's a generic class of type T. So in our case, that T would be product because we're sorting products. So this is going to compare one product and another product and return an int. And if the int is a negative number, then the first argument is less than the second argument in the sort order. If you return 0, it means they're both the same in your sort order. And if it's greater, then you need to return a positive number. So, how do we implement that comparator in our code? Well, we've got products by weight, and this returns what we've got here, a Comparator.comparingInt comparator. So again, that was a Java 8 implementation of a comparator, and it's got really nice static methods on the interface called comparing and comparingInt if you're using the int specialized field to compare on. And those take functions like Product::getWeight that extract values from the product. So this will build your comparator that says, take that getWeight method, apply it to the first argument, apply it to the second argument, take the resulting weights, and compare them. We can actually implement in a more old‑fashioned way as well here. So I'm going to show you how to use the old fashioned way just in case you're stuck on a pre‑Java 8 project. So for that, we would want to take our first product, get the weight, and we want to take our second product and get its weight, and we're going to compare those weights. So there's actually a nice method called Integer.compare that can be used to compare those two products. Now under the hood, this is what Comparator.comparingInt is actually doing, how it actually works. It pulls out to getWeight, and then it uses the Integer.compare method to compare those two values, or at least applies the same logic as Integer.compare. But, I have no time for old‑fashioned code now. Java 8 has been out long enough, so we can use comparing and comparingInt.

### **Shipments Example 3 (Live Coding)**

And now, let's take a look at the concluding part of the shipments example. I'm going to find the split point, so I'm going to create a method called findSplitPoint, and that is going to return the integer, which is going to be the index for that split point. So, we're going to take the products.size, and then I'm going to use a for loop with indexes. And the reason why I want to use indexes here is because we're trying to find the index, the split point, within that list. So, firstly, we need to get the product at the given index. Then if the weight of the product is greater than what we've got here is our LIGHT\_VAN\_MAX\_WEIGHT, so that is a maximum weight that we can fit for a product in our light van. If our product weight is above that, so that would be the first index of the heavy van, then we return the index in question. So now our split point is the first position in our products list of a product that needs to go into the heavy van, and we need to create our two subviews of this list using our subList method. Though, firstly, we're going to create the subList for the light van. That's from 0 to the splitPoint, remember, we said subList was end index exclusive, and then we want to say products.subList splitPoint and products.size. Finally, we need to create some fields to store these results in, so a List of Product here, which we're going to call our lightVanProducts, and another List of Product called our heavyVanProducts. Our lightVanProducts is the first subList, and our heavyVanProducts is the second subList. And then we've got getters here down below in the class where we need to return the heavyVanProducts and the lightVanProducts as well. If we pop back to our ShipmentTest and we run this code, we see the test passes, but we also want to make sure that we haven't broken anything else with our code. So let's run all of the tests just a good point of practice, and they are all nice and passing now, so that is good. Okay, so, what have we learned in this live coding example? Well, firstly, we learned to encapsulate collections, so put them within your business domain class, don't let collections code run rampant throughout your project. It's really useful, but you don't know when your requirements are going to change unless you don't know when your collection type is going to change. Secondly, we used the simple add method, and then we used the indexOf and the set method in order to implement replacing a product. We implemented the prepare here, so we used the sort method on list, we found the splitPoint using a for loop that used indexes over our list, and then we used the two subLists. If we just wanted to iterate over our list, for example, printing out, we can use the for each loop. We wouldn't need to use this legacy style of loop. We only did that because we wanted the index. And finally, we saw these views, the subLists, that let us see subsections of the list that we were going to play around with. Fantastic! And now, you know how to use the List API.

### **Implementations**

In this section, we're going to talk about implementations. Now I touched briefly before on the fact that our interfaces with collections define the functional behavioral characteristics, and we have different implementation classes that have different performance characteristics. This is the case with lists. We have a list interface, and we have an ArrayList and a LinkedList that implement that list interface. Those are the two implementations we're going to be talking about in this module. Those the two implementations we're going to be talking about in this section. There are other implementations of list in the JDK. So, there are concurrent implementations such as CopyOnWriteArrayList, but those are out of the scope of this course. You can also implement the list interface yourself, and it's pretty easy to do so by extending the AbstractList class. So the ArrayList is our first implementation of the list interface, and an ArrayList is a list implementation that has a backing array. Now, this is fantastic from a performance point of view most of the time because if you just want to get an element or read an element, you can just read it out of that index in the array. Very, very efficient and very, very easy to get that element out. Most the time when you want to add an element it's similarly cheap. You look for the slot that is available in the array, write your element in, and, hey, presto, you're done. Very nice and fast and very nice and cheap. Unfortunately, adds have a kind of corner case or a less frequent case that you hit. That is to say, when you run out of slots that are free in your backing array, and that's when growth happens with an ArrayList. The way the ArrayList grows is that you can provide an initial capacity to your ArrayList that defines what the backing array starts off with. But, most of the time, you will not do this. Most people will just initialize in the ArrayList and let the JDK deal with it. In that case, you'll start off with an empty backing array to begin with. And when you add your first element, it grows the default initial collection size, so that would be 10. Once you're out of your 10 elements in your initial ArrayList, it starts doubling in size. Now, an alternative strategy might be to just add one to the size with the ArrayList every time you add an element. That would minimize the amount of overhead you need for additional memory for this backing ArrayList. But unfortunately, the act of growing an array takes time. You have to allocate a new array of elements, you have to copy over the old elements, and then you can just put your single new element in place. So, there's always a tradeoff there between trying to find a growth strategy that minimizes the unnecessary memory overhead of having unused free array slots and also minimizes the amount of unnecessary copying when you have to grow your ArrayList. And doubling in size has proved to be an effective tradeoff between those two things. The ArrayList is a great default implementation choice to use when implementing a list. It's a very good general purpose implementation. It works well on a variety of different scenarios and probably should be preferred to a LinkedList most the time. The reason behind this is it's quite sympathetic to your CPU cache. Now, what do I mean by that? Well, modern CPUs tend to operate much faster than main memory. So, for example, loading a value from main memory may take unmodern hardware to 300‑CPU cycles, whereas adding up two numbers will just take one. So, a lot of the time the process of optimizing or writing code that's very fast is about writing code that is sympathetic to your CPU cache, and arrays are very sympathetic. As you stride through those arrays, you're reading through memory linearly, and your CPU cache is very good at prefetching linear memory accesses. It realizes the next value is the next cache line to be read and starts pulling those into main memory as soon as it possibly can. Other data structures, we'll look at LinkedList later, tend to have pointers going off between different Java objects, and that results in very, very hard‑to‑predict memory accesses, which often end up taking a lot longer and will slow your system down. Our other non‑concurrent list implementation choice in Java is a LinkedList. Now, LinkedLists have head nodes, they have tail nodes. Those could be the same node if there's only one element. And between each element and its successor element, there is a pointer to say what the next element is. LinkedLists in Java are doubly‑linked lists, so they actually have pointers back from each element to its predecessor, as well as its next element. Each element then has a pointer from the LinkedList node to the element itself. So, for example, in our three‑element LinkedList, we've got a head element, a pointer to its door, so that's the element stored at the head. The head has the next pointer to the middle node, which also has a pointer to a panel, and it has a pointer, then to the tail element, which also has a pointer to the window. And obviously, the tail has a pointer to its predecessor, and the middle element has a pointer back to the head. So, that's how LinkedLists are structured in memory. The result of this is that whenever you have an operation over a LinkedList, you tend to do a lot of pointer chasing. If I were to try and find the second element in this list, I would look at the head element, I would chase the pointer to the middle element, I would chase the pointer to the tail, and then I would chase the pointer from the tail to the window in order to look it up. There's quite a lot of different work that's going on here, whereas with arrays, we can just index to an offset in memory and then follow the pointer to the element. This means that LinkedLists are way less CPU cache friendly than ArrayLists, and in fact, as a result, I would consider them to be worse performance in most cases. I don't want to say in every use case they're worse, but they are worse in most cases. LinkedLists also implement the queue interface. They do provide certain functionality that the ArrayList implementation doesn't. And there are certain situations where LinkedLists I have found to be useful in my career in terms of being performance wins. So, for example, when adding elements at the start, LinkedLists can be good if you repeatedly add elements to the start with an ArrayList, then you tend to result in copying a lot of array elements to shuffle them further along the array. And if you're adding and removing a lot, there are certain situations where LinkedLists wins. But, even in these situations, you can sometimes produce benchmarks where LinkedLists lose out to ArrayList. So it very much depends upon your specific use case that you might encounter. So, in general, if you want to try and performance tune your software and optimize it, the best approach is to profile your production system with continuous production profile, something like Opsian or Java Mission Control, and benchmark your actual code. I'm just trying to provide here a performance characteristic of the implementation of these different data structures. They offer you some general principles as to which approach is best for your use case. So, these aren't hard and fast rules. These are just saying, if you have data access patterns that looked like this, you should probably use this thing. These are provided in terms of Big O notation, so Big O notation indicates how the operations performance scales with the size of the list. That's what the N in these cases means. It's the size of the list. O(1) means its constant time, so it doesn't matter whether your list has 100 elements or 1,000. It'll take roughly the same time. O(N) means it grows with the size of the list. And, you might see Omega 1 on the ArrayList add operation. That means that there's an amortized costs that's constant. I'll get to that when we hit that element in question. So, firstly, let's take a look at the get operation. For ArrayList, this is an O(1) operation. You just look up the element from the backing array using its index, and then return the value, so constant time. With LinkedList, if you want to get to a certain element, then you have to go through every single element. You have to chase the pointer in the list in order to find that element number N, so it's O(N). For adding elements, ArrayList worst case is O(N) because you have to copy all the elements and expand the list. But I say it's amortized constant time because over time, as you get a larger ArrayList that you need to do more copying with, you have to do it less and less frequently, given the number of elements that you add because you only do this operation when you have to double in size. For LinkedList, adding is O(1) because you can just add to the tail elements, so that's nice and cheap. If you have the indexed version of the add operation, obviously that would be O(N) for LinkedList. For contains, in both cases, the worst case is searching through the entire list to see if an element is there. So it's O(N) in both cases. For moving to the next element with an iterator, its O(1) on the ArrayList because you just increment your index within the iterator, and then you can look up the next element based upon the index, and its O(1) with LinkedList because you have to just chase a single pointer to get to the next element. Remove is also O(N) in both cases. So, for ArrayList, the worst case is you have to remove, say, the head element of the ArrayList, the first element, and then you have to shuffle everything up after it. For LinkedList, you have to go through to the given index for the remove that you're going to try and remove at, so that could be an O(N) operation because it's the worst case there for trying to find that remove. And then you have to do the removal. But once you've unlinked that element to remove it, it's a very simple operation. Just connect the predecessor and successor elements together to actually remove the node in the LinkedList, so much better.

### **Conclusions**

Let's summarize some of the key conclusions that we've seen during the course of this module. So, we demonstrated the key features of the List collection. You've now seen an example of a concrete type of collection with the List and how it can be used in practice with our shipments example, a more real‑world example. We looked at the different performance tradeoffs involved in different implementations. This is a common thread when we look at all the different data structure implementations within the different collections. Different approaches perform the same overall task, but they have different tradeoffs in terms of speed and in terms of memory consumption as well. Finally, I think it's just worth saying lists are really commonly used. They are the most commonly used collection that you will encounter, and you'll see them all over the place in Java code, so being an expert in lists really pays dividends. Up next, we're going to have a look at the map collection, so that's a collection with key‑value pairs, and that is also very commonly used and very powerful and another core Java feature that's really useful to know.

## **Collections of Pairs: Maps**

### **Introduction**

Hello and welcome to this module, which is on maps. Maps are collections of pairs, so they have keys and values in their collections, not just individual values like we saw with lists. If you're thinking about a real‑world object that's quite similar, you could think of a dictionary. Dictionaries have words and then either a definition associated with the word or, like this dictionary, the same word in a different language. Now you might ask yourself why, especially if you're coming for a .NET background, did Java use the more mathematical term map rather than dictionary for its key value association. And the answer here is actually that dictionary is a class in the JDK, and it predates the original Java Collections API and is kind of deprecated at this point in time. So don't use the dictionary class. Use the map interface and its subclasses. Within Java maps, keys are unique. The uniqueness is defined by the keys equals method, and as we'll see in this module, you might need to implement the hash code method or a comparator in order to get the map to function properly. Values don't have to be unique within a map, but they often are. In this module, we're going to see a variety of different things. Firstly, we're going to talk about what are the use cases where we might want to actually use a map, not just needing things like list and set values, but having this key value mapping. Then, we're going to talk about views over maps. We saw in the previous module, you can have views over lists, like a sublist. You can do similar things with maps. They're very useful things that you would use if you're using a map. Then there's the Java 8 enhancements. So Java 8 enhanced the Map API quite a lot, and it's worth talking about. Then we're going to talk about the different implementations of map under the hood. HashMap and TreeMap work very differently, and it's interesting and important to know the distinction there. Finally, we're going to talk about how you correctly use HashMap, the most popular kind of map.

### **Why Use a Map? (Live Coding)**

So in this section, we're going to talk about why we might want to use a map to begin with. And we're going to do that by implementing a certain interface in both an array list and with a map, and we're going to see the map is faster and uses less code. Here's our interface. It's called the ProductLookupTable. We can look up products given an ID. So a customer might come into your store, they might say, hey, here's the ID from the catalog. Can you give me some product details? What its name, what's its weight, and we ship them. We can add products to catalogs, and we can clear our ProductLookupTable as well. So we're going to look up our product given an ID. So if we implement that, we need to look through with a for loop for each of the products in our products array. If the product's ID is equal to the ID that we want, then we're going to return that product. And at the end, we're just going to return null. And the other thing that we need to implement here is our addProduct method. Now that has a very similar structure. We're going to loop over products, but we want to make sure that we don't have duplicate IDs here. So we're going to take productToAdd.getId, productToAdd.getId. And if we find a duplicate ID here, we're going to throw a new IllegalArgumentException, and that is going to say duplicate product, unable to add id for and then productToAdd. And if we don't have any duplicate products, we'll just take the productToAdd. And we'll take the products array list, and we'll just use the add method from and array and add it in there. Finally, we've got a clear method that we're going to take products and clear that backing array list. So the clear removes all the elements. So the problem with this code is it looks pretty ugly if we're being honest. It's quite bulky, quite verbose. It's quite inefficient as well. We have to search through all of our products in order to implement it. So a better implementation would be using a MapProductLookupTable. So we've got here our MapProductLookupTable, so this is our map‑based implementation. Then we're going to kick their off by initializing a map from keys to values. So the keys here are the IDs of the products, and the values are the product. And we're going to use the HashMap implementation. Later on in this module, we'll talk about different implementation characteristics. But for now, we'll just deal with it like this. I follow the naming convention of key name to value name, so idToProduct is my map here. So, firstly, we want a check about containsKey, and we want to call productToAdd.getId so that ID is going to be our key. And we want to check if it already contains the same product with that key in. We're going to throw our IllegalArgumentException that we saw earlier, Unable to have a product duplicate id for. That's the productToAdd. Nothing jubbly. And then at the end, we will just take productToAdd, and we're going to put that into our idToProduct map. So to add things in, we need to use that put method. Our id is the key, and our productToAdd is the value. Nice and simple. We also want to look up a product given an ID. So here, we have a very simple method that we could just delegate to. So we could say idToProduct.get(id). So just call the methods to look up the ID on the idToProduct map. Very, very simple. Just calling the get there. And at then at the end, we need to implement the idToProduct.clear, which is a very simple implementation. Well that's pretty similar to the one in the naïve one. So this is a much easier‑to‑read code, much easier‑to‑write code. We've removed a bunch of loops. We've also removed a bunch of inefficiency. So I've got this class called the LookupTableComparison class. This is not a rigorous scientific benchmark and should not be treated as such. It's just an example of how you can differentiate and the amount of speed difference that you might get between these different implementations. So this code generates a bunch of random products, then it adds those products in, and then it looks the products up afterwards. And we run that with our map‑based implementation and also are NaïveProductLookup implementation. So the differences in speed between the five runs here are due to warm up from the Just‑in‑Time compiler in Hotspot. NaïveProductLookupTable, 3 times faster at the end in the first run. But even there, it's taking 700 ms in order to look up items. Here with our map‑based implantation, it takes 9 ms. So about 70 times faster on this test with maps and array lists. That's a real win and something you should try and take advantage of. But the main win here is that maps are producing nicer, easier‑to‑read code.

### **Map API**

Okay, let's take a look at the Map API. Firstly, how do we add and replace values? Here's the put method. The put method takes a key and value and puts it into the map, associating that key with the value. Here, K and V are the key and value types within the map. Remember, map is a generic interface and can have keys and values of different types. You can see the put method returns a V, and that V is the previous value that used to be associated with the key in the map. If there was no value there, it'll return null. If you want to have a bulk update taking another map and putting all the elements in, that's what the putAll method comes from. I don't want to go into explaining what the ? extends K or the ? extends V generics constraints are. If you want to take a look at those, then I would suggest you look at my Java Fundamentals: Generics course that explains this kind of stuff in loads of detail. As to whether you can use nulls for keys and values, that is a characteristic of the map implementation. So HashMap will allow you to have a null key and null values. TreeMap will allow you to have null values, but no null key. And if you have other third‑party implementations, they will usually define whether they allow or disallow null keys and values within their Java doc. Having added values into your map, the next question is, how do we look up values in the map? How do we pull them out? And the obvious answer is the get method. Get returns your value, your V, and it takes a key as an object. So it will look up that value. If the value is missing, get will return null. It won't throw an exception. It won't fail fast. It will just return null. ContainsKey and containsValue are the methods for looking at whether a key or value is within the map. If you want to know whether a key is within a map, and then you want to use it, I strongly recommend that you call the get method once and then just check whether the return value is null or not. You will save having to look up that value twice in the map. Now it's also worth noting that even though containsKey and containsValue are very similar in terms of naming and API, they have quite different underlying performance characteristics. ContainsKey and Get use the maps implementation, the fact that they're fast for looking up values from keys to respond quickly. See the section on implementation for different implementation performance characteristics. ContainsValue you would expect in most cases to be slower than containsKey because it might well have to search through all the values and certain implementations in order to find whether the element is in there. The final thing to note about this API is that get, containsKey, and containsValue all used the object type rather than the K key generic. And that's to allow you to have a bit more flexibility with using this code. Suppose you've got a map with strings that are keys, and you have a local variable that is of the type object and it has a string in it. By allowing object to be used in the get or that containsKey lookup, it allows you to use that variable without introducing an additional cast to a specific key type. Even if you added the cast, you would still not achieve complete safety. Here get if the element is the wrong type, will return null. ContainsKey will return false. There aren't additional runtime errors that get introduced as long as you're using the Map API correctly. And finally, removing. If you remove an object by key, just call the remove method. Nice and simple. The V that gets returned is the old value that's associated with that key or null if it wasn't present. And again, if you want to remove a value and check whether the key is contained there, I would only call the remove and check the return value for null rather than doing a contains key and then a remove. Clear, as we saw in the previous demo, provides a bulk update operation for clearing out all the values just as you might see in the Collection API. Querying size also has the exact same semantics as with collections, size to find out the number of elements, and an isEmpty method that shortcuts whether the map is empty or not and potentially may work faster than checking whether size is zero. Collection and Map. There's a bit of a strange API interaction here. Map doesn't implement the Collection interface unlike the other collections that we see in this course. That doesn't mean it's not a collection. It's still part of the Collections API. It just means that because it's dealing with keys and values, the contract of the Collection interface doesn't quite fit for map.

### **Views over Maps (Live Coding)**

So, in this module, we're going to look at Views Over Maps. Just like lists in the previous module, maps can also have views, and this module is just going to be live coding through a little bit of an example to show you what the views are like on the Map interface, and how and why you might use them. So we first of all here got our idToProduct map that has a door, a floorPanel, and a window in with the product IDs of 1, 2, and 3 that we've added here. The first view we're going to look at is the keySet. So if we call keySet, that returns us a view of the type Set<Integer>. Now, we haven't actually looked at Sets yet, because the Sets module comes later in this course, but a Set is essentially a collection where each of the objects are unique, so there's only one element in the Set. The keySet here is the Set<Integer>, so we're going to print out the ids set, and then I'm going to print out the idToProduct map. And if we run that code, and we can see that we've got our keySet of 1, 2, 3 and then the map with the three different products in. Now, just like the sublist views with lists, we can remove elements from views. So if we rerun this code here, we've added the remove on the first product, and we can see that the view only has 2 and 3, but also the backing map here has had the Wooden Door removed from it as well. Something that's important to be aware of, though, is that even though views modify their backing collections, we can't add a new element to the view on the map. Now it says UnsupportedOperationException. The reason here you can't add the extra key into the map is because in the map you need the key and the value. So you need both the key and the value, and here by adding just to the keySet, you're only adding a key, and there's no value associated with it. So another view we can take is the values view. So this is our product within the map. This just returns the Collection interface, so it's a collection of products. Again, we can take that product, and if we look at the ProductFixtures, which is where we got these example products from, and we can take our window and remove that from the products map. We print out the products here. I'll print out the idToProduct beforehand. Remove the window, print out the products, print out idToProduct after that operation, and again, we can see that we were able to remove the value here, and now the only thing that's left is the Floor Panel. But just like the keySet map, we cannot add in the window to the products collection, because we're going to get the same UnsupportedOperationException, because, again, you can't add a value without a key, just like you couldn't add a key without a value. There's a third view within Maps which is very interesting, and that's called the entrySet view. Now, unfortunately, entrySet has an absolutely horrific type parameter that is Set<Map.Entry<Integer, Product>> entries. Wow, what does all this stuff here mean? Well, it means we've got a Set, and that Set contains entry elements, and those entry elements are basically a pair. Java doesn't have a pair type, so we have this Map.Entry interface, the entries implement, and it has an entry of Integer, an entry of Product, and those are our entry values. Now if you're programming this on Java 10 or later, you can use the var keyword. That is strongly recommended for things like this to absolutely cut down on that generics boilerplate that can be hard to understand. That's the local variable type inference. If you're on older versions, you need to put up with this approach. Now, what we normally do with those entries, or what a common operation is, is to loop over them because it's a Set, so it's got a collection so we can loop over it. So I'm going to just print out the entry here, and I'm also going to print out the entry.getKey, I'm going to print out the entry.getValue, and we'll see what we've got here. So we've got the key is 2; the value is the product, the Floor Panel; and the Map.Entry has that key equals value to String, so it's quite debuggable, quite easy to read. Now, entries are themselves mutable, and we mutate the view. So I'm going to set the value here, and I'm going to set it to ProductFixtures.window, and then at the end I'm going to print out the overall map, and we'll see that the entry has been set to be a window for product 2, so this is a useful feature that we have with the entry interface that we can modify that value. Another thing that's interesting is that we can actually create an entry value. So we could here, for example, create an entry with an id of 3. We could have the window ProductFixture here, and that would be an entry value, and we can try to take that entry and add it into our entrySet. This Map.entry method was only added in Java 9, so that is something to be worth noting. If you're using Java before Java 9, the Guava library has an implementation. Unfortunately, we cannot add this entry into the entries Set. So there we go, the three ViewsOverMap: keySet, values, and entrySet.

### **Java 8 Enhancements**

We're now going to talk about the enhancements to the map interface in Java 8, which are actually quite extensive. If you're not using Java 8 in your current system, it's strongly advised you upgrade to it, it's been out for several years now and is very stable and useful. What you get with Java 8 is a whole bag of useful and fun functionality that massively simplifies a bunch of common programming tasks. So first of all, let's take a look at adding and removing values. There has been a replace method that's been introduced that lets you update a single value. So the difference between this and the put is the put will add in a key and value pair into the map, even if the key isn't there previously, whereas with replace, if the key isn't there, it won't make a change. ReplaceAll takes a BiFunction, so that will take a function that takes a key and the old value, and returns a new value, and you apply that function to every key value pair in the map. Remove removes a key only if it has a value. So, remove normally takes a key, and so it will look that key up, and it will remove it no matter what the value is. This remove method only removes the key if the value is equal to the value passed as a parameter. We will be looking in more detail at the example methods we have here in the live coding section. Next, let's take a look at methods that update the map. Firstly, there's getOrDefault. So regular get will return null if you have a missing element, whereas getOrDefault takes a default value that will be returned in that case. This can often simplify null checking and reduce the likelihood of null pointer exceptions, a dreaded Java programming issue. ComputeIfAbsent takes a function that lets you compute a value if it isn't there. So you'll give it a key, it'll look up the value. If the value is there, it will return the value for that key, if not, it will call your function to compute the absent value. This new value will also replace in the map. PutIfAbsent puts a value only if it isn't present in the map already. ComputeIfPresent computes a value if it's present. It takes a key, it looks up the value, if the value is present, it replaces it with the new value, if not, it makes no mutation to the map. Compute is kind of like the generalization of computeIfAbsent or computeIfPresent. You look up a key, it looks up the value, takes a function. If the value is present, it will give you the value, if not, your computation function will take a null value, and you can update it or introduce a new value. And finally, merge. Merge takes a key and a value, and the remapping function, and if there is no value associated with that key in the map, the value is used as the new value, put into the map, and returned. If there is an existing value, the merging function is called on both the existing value and the new value. So that's really useful for example, trying to keep something like a map of totals, for example, and then update it so you can pass in one as the new value and have addition as your merging function. Finally, there is forEach. ForEach is a convenience‑based callback for iteration, very similar forEach on the collection interface. Here, forEach is made super useful by the fact that there is not an easy way with maps to necessarily iterate over the key value pairs. Normally, you have to use the entrySet method and then call an iterator on the entrySet, and then deal with entry of key value elements. ForEach just passes the key and value as two parameters to your callback function.

### **Java 8 Demo (Live Coding)**

Okay, so let's take a look at the enhancements in Java 8 from a live coding perspective, a demo perspective. We've got the software demo here, we've got a few products in our HashMap. So door, floorPanel, and window again, product IDs 1, 2, and 3; and I've also got at the top, a default prod product, so, that's just a default value that we can put in there. Now, as we've talked about with products, if you look up a product by ID, and the ID is missing, then, we will get a null value, and I could also write that like that. If I run this code, we can see we've got no product four, so we're going to get the null value. But it's often the case with these kind of situations that you want to have a way of providing an alternative, so, you'll get that product here, go, you know, if the product's null, return this default instead. And Java 8 added this very helpful convenient getOrDefault value, so, we can pass the number four in here, and we can pass the default product as well. If we rerun that, we can see our product got this default product a negative ID, so that's something that lets us differentiate from normal products. It didn't actually update the map here, as we can see from the fact that calling .get(4) still returns null the second time around. The next Java 8 enhancement that we're going to see is the replace method. So, we've got a replace here and we are going to try and replace product ID number one, so that's our door, and we're going to replace it with a new product that's also going to have an ID of one. It's going to be a Big Door, and then we'll have a weight of 50 here, and we can also see the products that we get returned. So that is the oldValue. So, I'm going to print out oldValue and then I'm also going to print out idToProduct.get(1). We can see as we run this code that our wooden door was the old value and our big door is the new value that we can see when we call get of one after the replace. So we've replaced our door with a big door. I'll show you another Java 8 enhancement, and that is the replaceAll method. So replaceAll has this super confusing and complex generic type parameter, BiFunction<? super Integer, ? super Product, ? extends Product>. So, don't worry about the super and extends and the question marks, if you really care about those, go and have a look at my Java 8 Generics course. The key point is what is this function doing? Well, it's a BiFunction, so it takes two arguments. The first is the key, so that would be the ID; the second would be the product, and I'm going to implement this BiFunction with a Lambda expression here. And what that does, it says, take the ID in the old product and return a new product, so replace each product in question, and then this function will be called for every pair of elements in our map. So our new product will have the ID of that, it'll have the name of every old product, and I'm going add 10 to their weight. So, if we print out, idToProduct beforehand and we print out idToProduct afterwards, I can actually choose to use the expression Lambda here rather than a statement style Lambda, we can see that all of our products have had their weights increased by 10. So that's a good way of performing a function that updates and replaces every single element within your map. That's the replaceAll function. Another super useful function here is the computeIfAbsent function. So, computeIfAbsent takes our idToProduct and we take our key, so, I'm going to use the key of four here, and our function, so that would be id to new\_Product. It's going to pass the ID in. I'm going to make this product a Door Handle, and have a weight of five. So I'm going to just pull this code onto the same line now. Lovely jubbly. So, what have we got here? Well, I'm going to print out the product, print out idToProducts.get(4), and we'll see what's happened. Unlike get or default, this uses a function to create the new value, and with computeIfAbsent, it will actually put that value that's being produced into the map as an update, but only if it's missing. So here we've got four, an ID not in the original map, but, if we choose to use a different key, so something that is in the map, we can see that it would not call that function, it would only apply the computation function if the value is actually absent. So, by choosing the key of three, we see that we get the old product fixture, which is the window, and not the new door handle. Finally, let's take a look at the forEach method. So the forEach method takes a BiConsumer, so that's a function that takes two arguments and returns void. The first argument is the id, and the second argument is the product. So remember, a couple of sections ago we went through the entrySet method? This is a much nicer, much better way of looping over the elements within the map than the old entrySet method. And here, we've got our two arguments, our id and our product, and we can print out things like prod.getName has an id of, and that's the id. If we run this, we'll see that printed out for our map for each of the values. Wooden Door has an id of 1, Floor Panel has and id of 2, Glass Window has an id of 3. It's a very useful way of looping over the elements, much more convenient and much easier to read than the old entrySet loop.

### **Implementations**

Now let's take a look at the different implementations of the map interface that we have. Now, just like the other interfaces, map has different implementations. They have different performance characteristics and tradeoffs that might be appropriate in different situations. These are implementations that are already done for you that ship with the JDK. You don't need to do any more work in order to use them, so you don't need to worry about all the details of how the algorithms work. But it is a good idea to have a basic overview of how they work so that you know which is the most appropriate implementation to choose in your development scenario. In this section, we're going to look a two general‑purpose implementations. Firstly, HashMap. If you're at all unsure, just use HashMap. It's a very good general‑purpose implementation. It's the only one of these two that will work if your keys don't have a defined sort order. It's very fast. It's very optimized. It's a good choice. TreeMap is an alternative approach that keeps elements in the sort order of their keys. That gives it additional functionality that lets you look at, for example, subsections of the map based upon that sort order, and it can perform better in certain circumstance. There are other implementations within the JDK, such as EnumMap, LinkedHashMap, and IdentityHashMap. These are not going to be covered in this section of the course. They're more advanced topics, but they're certainly things that might be worth reading in your own time. So HashMap, how does HashMap work? As we say, it's a good general‑purpose implementation. When you put a value into a HashMap, it takes the hash code of the key that you're adding, and it takes a bucket of elements behind that, a bucket of nodes. It computes the hash value from that key, modulus the count of the buckets that the capacity of this array, and that defines a slot within the backing array with which to store it. If you're going to do a lookup operation, you do the same, get the hash code, modulo the bucket count operation. Now, sometimes you might find multiple elements that have collisions. That is to say they're different keys that result in the same hash code. This is within the hash code method contract, but it's undesirable. It's undesirable because collisions result in buckets expanding. Each bucket becomes a linkedList initially. So instead of you having just do the hash, look up the element in the backing array, and then read the value back, you have to start searching through that linkedList to find which element is the correct element and checking the equals method on every key of that linkedList item. Now, in Java 8 or later, those buckets get converted to trees when there are more than 8 elements in the linked list. There's a tradeoff there between trees having more memory overhead and also scaling better when you have lots and lots of collisions. Now that's the key point. The number of buckets can also increase with more elements, so you can resize that backing array with the HashMap, a that's very useful. As you carried on adding elements to a fixed‑sized HashMap, you just get more and more and more collisions, and collisions are bad because they result in slow adding and lookup times. TreeMaps have keys with a defined sort order, either due to implementing the comparable interface or providing a comparator. Under the hood, they use a red/black binary tree, so that's a balanced binary tree. In a binary tree, unlike a linkedList, each note has two successor nodes, potentially, instead of just one , and the idea here is that each node contains an element, and nodes with lower elements in the sort order go to the left, and nodes with higher numbers in the sort order go to the right. The tree gets rebalanced so it never gets longer and logged to the base 2 of N in terms of depth, where N is the number of elements. In general, TreeMaps are slower than HashMaps because there's a lot of pointer chasing and comparison work done when you're using a TreeMap. But they often provide functionality that HashMap doesn't, so they implement the navigableMap and sortedMap interfaces. We won't cover the details of those, but they can be useful if you are looking at dealing with keys that have a sort order. Okay, let's compare the performance of our HashMap and TreeMap implementation. Firstly, putting an element for Hashmap, the worst case is that you have to re‑expand the HashMap and re‑expand the backing array. That means copying all the elements so it scales up in O(N) time, where N's the number elements. Mostly, though, you just look up the hash bucket based upon the hash key and replace the element in question, so most the time it's pretty constant amortized Ω(1). TreeMaps O(log(N)). The worst case of put is you have to go all the way down the tree to the bottom and the tree is capped in size O(log(N)), so that's number of times you need to jump through the tree. Get, very similar, exactly the same reason. The worst case for get is that you hash into a bucket in the tree, and you have to traverse the linkedList so that would be O(N), Java 8 or later, because there is now a tree that backs that linkedList, so in Java 8 or later that will be O(log(N)). And amortized again Ω(1), because most the time you just do the hash code lookup and pull the value out. For TreeMaps it's O(log(N)) again. The worst case is you have to go right to the bottom of the TreeMap, and the height of it is O(log(N)), where N is the number of elements. ContainsKey works basically the same way as get, except instead of returning the value, you just say whether the key is in the map, again, so it has basically the same kind of algorithmic properties. Next, so next here is when we're iterating through entrySet on the map or calling the for each method and we're going to the next element. The worst case with HashMap is capacity, so that's the size of the backing array divided by the number of elements. The worst case there is that you have to scan through the list, finding a gap in the backing array until you get to the next element. And then with TreeMaps, again, the newest case for next is you're at the bottom of the tree, and you have to go back up to the top and then back down again to the bottom in order to get to the next element in that sort order, and that's O(log(N)) because the height of the tree is O(log(N)).

### **Correctly Using HashMap**

Hello, and welcome to the final section of this module on maps. In this section, we're going to look at how you can break the HashMap contract by mutating the keys, and why you don't want to do that. And then we're going to summarize what we've learned in this module. Okay, so first of all, I've got an example class here called MutableHashMapKeys, and that class has another class within it called a MutableString. This MutableString has a single value that is settable, so you can update it, and it delegates its hashCode value to that set value, and that's what it delegates its equals to and what it delegates its hashCode method to. So, what we've got here is an example of using that code. So what we're going to do is we're going to take that MutableString class, we're going to change the key, and we will see that it breaks the HashMap. If we take our brokenMap, our MutableString to String map, so it's got MutableString keys and string values, and it's just going to be the same. If I run this in a debugger, we can step through the code and see how it works. If we step over this code, we've added in abc as our key. So if we use the key abc, we look that up in the map, call in the get method and we print out the whole map, we have a value of abc from our MutableString abc, and our map is equal to abc=abc. If we set the key's value string to be def, and then do the same operation, we break the HashMap. When we do a key lookup, it returns null, and when we look at what's in the value, it says def=abc. So the null is the real problem here. Changing that value within your key changes the hashCode of the object in question. And if you're going to use maps correctly, you always, always, always need to make sure that they return consistently the same hashCode when it's called again, and again, and again. By mutating that value, we've resulted in a hashCode that can be modified, and that means that when you call the hashCode within your HashMap, it hashes to the wrong bucket within the backing array, and then the wrong value comes out. Do not mutate the key within the HashMap. Make sure that its hashCode always returns the same value, and make sure that it is always continuing to be equal to itself. You want immutable keys in HashMaps. That's the safest thing to do. Okay, let's summarize this course and recap what we've learned. Maps associate keys and values. They're not like other collections that just have single element types. We saw two key implementations, HashMap and TreeMap. TreeMap can be used for certain circumstances, and has useful facilities, but HashMap is the general purpose default choice that you should probably use most of the time. The API is still improving in Java 8 and beyond. Later on, we'll look at collection factories that provide useful ways of building maps as well. Whatever you need, Java has you covered. There are a bunch of other implementations within the JDK that are very useful in certain specialized circumstances like EnumMap if your keys are enums, IdentityHashMap or, for example, LinkedHashMap. Good things to research at home. In the next module, we will be talking about another Java 8 improvement, which is streams.

## **Java Streams**

### **Introduction**

Welcome to this section on Java streams. Streams are a powerful new abstraction introduced in Java 8. They provide a way of performing aggregate operations over entire collections at once. This is an alternative to the traditional approach of using foreach loops or iterators to operate on collections. This section is just a short introduction to streams. If you want more detailed coverage, and I've written a book called Java 8 Lambdas that covers this topic, and Pluralsight also has a video course dedicated to Java streams. So what's the problem with for each loops and iterators? At the end of the day, they're a low‑level programming construct. Every time you use them, you have to write a lot of boilerplate code. It's often simple code, but potentially very error prone and a real waste of time. With Java 8, a simpler solution was introduced. Streams are that solution. They offer a way to do functional‑style programming in Java. This means that you write your business logic in compact functions using lambda expressions or method references and apply operations over entire streams of values rather than having to mainly write loops and then build the collections yourself and write a lot more logic in order to do common data processing operations. This section provides a brief introduction to streams. So, firstly, we're going to look at how streams can be used in a little bit of Java code using our suppliers and products examples. Then we're going to go through a catalog of different stream operations so you can see what options you have when using streams. Then we'll look at collectors. Collectors are an addendum to the streams framework that allow you to build final values. They're very, very commonly used for building collections themselves like lists or maps. Then at the end, we'll briefly go through the limitations and general approach of the streams API so you know when not to use them, as well as when they are a good fit.

### **Live Coding Streams**

Hello, and welcome to the live coding section of this module. Here, we're going to look at how streams can be used in order to solve some relevant business problems with products. And we're going to give you a simple example of the streams API and show you how it compares with using loops and iterators, the more traditional approach. So, here is our example problem. We've got our three products, our Wooden Door, the Floor Panel, and the Glass Window. We're going to add them all into an array list of products, the door, the floorPanel, and the window, and then another floorPanel and another window. We then want to calculate the namesOfLightProductsWeightSorted, and it's the Loop version of the code here that we've got to begin with. We'll see that we've got our list of products that we take in. Firstly, we want to take the products that are below 30 in weight. We're going to add them into our lightProducts list. Now once we've found our lightProducts, we can sort those products, comparing them by their weight. And then at the end, we are going to loop over those lightProducts list, printing out their names. So we're printing out the names of the lightProducts that are sorted by weight. Now, the problem with this code is you can vaguely see what's going on here from the name of the method. And if you walk through it step by step, you can see what's going on in detail. But we've had to do two different loops to solve this. It's not very clear exactly what's going on. You can't skim it. It doesn't really read like the problem statement. And that's the benefit that streams are already providing for us, reading like the problem statement, the ability to have code that reads a lot like the business problem that we're solving. I'm going to run this code here, and we can see what it outputs. It does actually work. It shows the Glass Window, Glass Window, Floor Panel, Floor Panel. Okay, so, let's take that code and implement it with Streams to give you an idea of how to use the streams API. Now the first thing with streams is how do you create a stream? Well, we're going to take those products, and it's very, very simple. We just call .stream on the collection object. And if we see what this method returns, it returns a stream of product. The stream itself is an interface with lots of different methods on it. I'll go through a collection of the important methods within this API in the next section. But for now, we shall just see how we can solve an example business problem using the streams API. So our first step in our loop version was to loop over the products and find products that had a weight of less than 30. That's what this line here is doing. Streams have a nice built‑in convenience method called filter, and filter takes a predicate. A predicate is a method that will let you remove elements that don't match a certain criteria. So, the function has to return true to keep it within the list, false to remove it. What I've done here is use a lambda expression to implement it. I don't want to go into the details of a lambda expression, but a lambda expression is just a function. So, this is a list of arguments. So that's like saying Product product, and then we have this arrow in the middle, and this is the body of the lambda expression. So here it is just an expression itself. It gets evaluated, and it revaluates to a boolean, and that's the value that gets returned from the function. I don't have to specify the type with a lambda expression. I can just use it locally like that. I can just use the type inference feature of it. If you want to learn more about lambda expressions, again, I've written a book, Java 8 Lambdas, or there is also the Pluralsight Lambda Expressions and Streams course for looking at this stuff in more detail. The next step is that we want to sort the elements in the list, and we've provided this comparator here using the comparingInt static factory method from the Comparator interface that was also added in Java 8. That was the same one that we covered in the previous section. The equivalent of doing sort with streams is called sorted, and we can provide the comparator to define the sort order there. Now what you'll note about these methods like filter and sorted is that they return stream themselves, so that's a stream of product returning a stream of product returning a stream of product. And that means that you can daisy chain these methods as you can see here with this dot, dot, dot style like you would with a builder pattern. Once you chain them, you can then perform an operation. The final step here is to get the name out of the products. We can do that with the mapping operation. We need a method reference here, Product::getName. And we'll see that map now returns a stream of string because those names are strings. And finally, in our operation, we're going to have a terminal operation. So that's the same operation that doesn't return a stream that just prints things out. And, in fact, IntelliJ is recommending here that we can use a method reference throughout the lambda expression. These :: are method references. So they say, effectively use a reference to this method in question. So don't call it immediately. Now it's a common part of the stream API here. We call the filter operation. This function is applied to all the elements in the stream. We call the map. This function is applied to all the elements the elements in the stream. We call the foreach. This function is applied to all the elements in the stream. These aggregate operations, a bit like you might see with something like SQL rather than more traditional Java. Now if we run the code again, we can see, again we get Glass Window, Glass Window and Floor Panel, Floor Panel. This code here might be more familiar code to you, but it is much harder to read what's actually going on. Whereas here, namesOfLightProductsWeightSortedStreams method here has code that says, take the stream of products, filter it, making sure that the weight is below 30, sort it, comparing on the weight, map each product into their name, and for each of those names, print them out. That is code that reads much more like the problem statement at a business level by level code. This is the future of Java. This is the direction that we're going in. Once you get familiar with it, it's a real win and absolutely fantastic!

### **Operations on Streams**

So now you've seen streams in action in an IDE, let's cover a reference of the different stream operations so you know what options you have and when you can use them. Filter removes elements from a stream that don't match a certain predicate. A predicate is a function that takes an element within the stream, and then it returns true or false, depending upon whether you want to retain it within the filter operation. In our example, we can see that we've got a stream of products, and we filter those products, checking their weight is over 20. And if it's over 20, they get retained in the stream. If it's 20 or below, they are removed. Map transforms one element into a new element for every element in the stream. So this is a common pattern whereby you might loop over elements in your collection, apply a function in your loop, and then return the resulting elements at the end. Our example code here, we've taken a stream of products and mapped each product into their name. The construct here is called a method reference, so that's like a lambda expression that takes the product and calls the getName method on it. Next we have the match family. These are methods that take a predicate and tell you whether elements within the stream match that predicate. In our code example here, again, we're using a predicate that checks whether the product's weight is over 20. Any match returns true if any product matches the predicate in question, so if there's any product for which the predicate returns true. There are two other variants of match. NoneMatch returns true if no products match the certain predicated applied, and allMatch returns true if every single product matches the predicate provided. All these operations are terminal operations, unlike map and filter. They return a Boolean value. Again, a predicate is used here similarly to the filter operation. Skip and limit are a pair of operations that are to do with discarding a certain number of elements. So skip discards the next N elements, where N is its provided argument it takes along. Limit only retains the next n elements and throws away anything after that point in the stream. A good example of how skip and limit are used is in pagination, and that's what this code example here demonstrates. Skip the number of elements on the page multiplied by the page number, so that gets you up to a given page, and limit only at the number of elements on the page. There are two variants of the sorted operation, both of which sort elements within the stream according to a provided order. Here is an example which takes no arguments and that just relies on the elements within the stream implementing the comparable interface. We've taken our products and mapped them into their name in order to get strings within our streams. Strings are comparable and have an alphabetically defined sort order. So if we called sorted, we will get those strings sorted alphabetically. Another example could be to take those products and just sort the product objects themselves by name. In order to do so, we would need to provide a comparator ourselves because products don't implement comparable, and we could just use a byName comparator, so just comparing each product by its name. I've provided a Java 8 example using that comparator.comparing method that we've seen elsewhere in this course. FlatMap is a more advanced variant of map. Instead of mapping one element in a stream to one element in the resulting stream, FlatMap is a transformation operation that goes from one value into 0, 1, or many values. So here we've taken a stream with our shipment objects, and we've flatmapped each shipment into their light van products and then called .stream on them because the flatmapping function needs a stream of values that it can operate on. So here we replace a stream of shipments with a stream of products, where all the products that can fit in the light vans within those shipments are the products in the resulting stream. There's several other operations that I'm going to go through now. Firstly, min and max. These provide the minimum and maximum element within a stream. Obviously, you need to define the sort order by which the elements are min or max, and again we can use a Java comparator in order to do that. If you want to just provide a side‑effecting operation for each element in the stream, such as printing out its name, saving it into a database, updating some state elsewhere in your system, for each is an operation on the stream and it just takes a callback for each element. That will be a consumer of T, where T is the element within the stream. Here we're printing out the name of products. FindFirst and FindAny get individual elements out of the stream, so just one single value from the stream. FindFirst, guaranteed to get the first element out of the encounter order If there is one. Commonly findFirst and findAny are combined with other operations within the stream API in order to find an element in question. So in our example here, we found the first product whose name contains a string chair and returned it. And finally count, counting the number of elements in the stream. So this performs a similar facility as size does on a collection, and it's often found to be used in conjunction with other stream operations. Here we've combined it with a filter, finding products whose name contains chair, so we're counting the number of chairs in our product stream. And finally, reduce. Reduce is a very general operation. You've definitely done a reduce before when operating on a list yourself. It's one of those operations where you take a collection of values and produce a single element together at the end by combining each value with an accumulator. In our code example here we're adding up the total weight of all the products within our stream. So 0 is the first argument to reduce, and that's your initial value for the accumulator. Our combining function takes an existing accumulator value, so in this case it would be an int, and the product, and that would be every product in your stream or every element in your stream, in the general case, and adds together the accumulator and the weight of the product. By combining those two values together, that becomes the new accumulator value at the end of the reduce operation.

### **Enter the Collector (Live Coding)**

In this section, we're going to look at collectors. Collectors are a part of the Stream API that allow us to build final values that are big and complex out of streams. For example, building collections and streams. And we're going to look at a few simple examples of how collectors can solve problems for you. Here is our EnterTheCollector class. Very similar to our stream live coding example earlier, we have a door, floorPanel, and window as input products. We have an ArrayList of products with, this time, an additional floorPanel in, so one door, two windows, three floor panels. And, at the end we've got a stream pipeline where we stream those values. We find the light products, we sort them by weight, and then we print them out. So, here we can see them printed out, we've got the Glass Window, the Glass Window, and then we've got three Floor Panels, nice and simple. But, what if instead of just printing these values out, we wanted to build a collection out of the product elements that are within that stream? So, for example, how do we build a list of product here? ForEach, as we can see, returns void, and that's why there is a compile error. We can't just assign the result of that sorted operation to the list because the sorted operation returns a stream and the list is a list. And this is where collectors come into play. Collector is a recipe for building a final value out of the contents of that stream. And in order to use the collector, we use a method on the Stream API called collect. Collect takes an argument that is a collector, and here we've applied the collectors.toList collector, but I ought to show you the signature of the method so you can see the collector as an argument. You can build your own collectors, collector is just the interface, but most the time you will end up using the collectors implementations that are shipped with the JDK. They are in the Collectors static factory class. You can see just scanning through this class that you can do all sorts of operations with collectors, and you can build different collections as well. Let's see what happens if we run this code and print out our list of light products at the end. Here we go. We get all of our products, and it's a product list, which we can see from the square brackets at the beginning and end. What about something a little bit more complicated? How about a map? We've seen those, how do we build a map? Well one of the most common operations of building a map is using the groupingBy collector. So groupingBy takes a classifier function that says the elements in the stream are going to be the values in your map, and what is the key that you want to associate with each element? Here, we could, for example, take the name of the product, and I'm going to change the type of our light products to reflect that. Collectors.groupingBy doesn't just result in a list of products, it results in a map from string to list of products. That's to say, a map where the keys are the names of the products, and the values are the list of products with that name. So, let's run that code again. So we can see the string, Glass Window, equates to our two glass windows, and the string, Floor Panel, equates to our three floor panel objects as well. We can actually go even further than that with collectors. Collectors are composable. That is to say that collectors can collect subsections of values within collectors. If I click through to see how this groupingBy collector is \_\_\_\_\_, it actually delegates to another groupingBy collector, passing in its classification function, and also the toList collector. And in fact, the list of product argument inside the map is constructed by that toList collector, and we can provide another downstream collector in order to use it. Don't worry if you're not following all the details here, I'm just trying to demonstrate how powerful this framework can be when you really get into it. So what I'm actually going to do is use the counting collector here. Now the counting collector, instead of adding the elements into a list, just counts the number of elements there. So we still have a map from string to list of products, we're going to get a map from string to long. A map from the name of our products to the count of the number of products. Here we go, Glass Window=2, Floor Panel=3. Very, very nice and simple. And can you just imagine how much code we would have had to write in order to get that result that we have with just this little section here, there's just this groupingBy Product::getName, counting collector. Collectors are really useful, they're really powerful, and they fit really nicely with collections. And I hope you enjoyed seeing this little introduction to them and now have the opportunity to go off and have a look at them in more detail.

### **Conclusion**

Let's wrap up what we've seen in this course and evaluate where the streams were a good idea or not in the context of your code. Firstly, we should ask ourselves, are streams always better than loops? There's going to be an extent to which this decision is always subjective and something that perhaps you need to think about yourself or in the context of your team members, but streams broadly have certain advantages over loops. For a start, they're a very high‑level construct. That means that you need to spend less time rewriting the same old boilerplate between every operation. Secondly, they're an optimized framework. So the JDK developers themselves put work in under the hood, making sure streams are relatively fast. You don't always need to worry about optimizing for that code. Then, there's the question of better readability. That's a subjective point. But streams are a higher‑level code construct, and they do tend to read more like the business problem statement, and in that sense, they are more readable, and that is a good thing. Certain corner cases do tend to be worse to read and worse to use with streams. Specifically, none of the streams functional interfaces throw checked exceptions, and the result of that is that you have business logic operations that do need to throw checked exceptions, they interact poorly with streams, and it can often be easier just to write loops. Finally, streams are only available in Java 8 or later. Most people nowadays are using Java 8, but not every deployment does, so you are restricted in that sense. Loops are a lower‑level construct. That can often be a downside, but it doesn't mean you have more control over what's happening. You can write any arbitrary operation that you can think of rather than just operations that are in the streams framework. And sometimes loops can be faster than their stream code equivalent. I wouldn't worry about this from most cases. But there are certain cases, for example, high frequency trading systems, where often that performance is very important. Readability is subjective, and some people do definitely prefer and find loops easy to read. In my experience, I would give streams a go, even though they can be unfamiliar and a little bit challenging to read at first. Once you're familiar with the constructs and the idioms, you'll learn to love them. Loops, as I say, work nicely with checked exceptions. Nothing else you can do about that. And often, it's just better to use a loop than to try and force that idiom into streams code. And finally, loops do work while on all Java versions, not just 8 or later. There's a lot of further learning material around streams. As I said, I wrote a whole book, Java 8 Lambdas, looking at the improvements in Java 8. There's also a series of courses on Pluralsight. For example, there's a course looking at using streams and lambda expressions that can be very useful if you want to learn this stuff in more detail and take several hours rather than just the 30 minute presentation I've provided here. Collectors are a very advanced topic and can be composed, and you can do all sorts of amazing, fantastic stuff with them. In this module, we've just shown you the introductory‑based cases, so collectors are definitely something that you can learn more about in the future. And finally, one of the other benefits of streams that we haven't really explored in detail is that they can run code in parallel. This can result in performance improvements. But, parallel streams aren't a guaranteed win, and your code can run slower sometimes, as well as faster. In summary, streams are a powerful new abstraction in Java 8. They can definitely replace many use cases of for loops and iterators, though, as we've said, they're not always the best thing to use, and you can, especially with things like checked exceptions, sometimes be better off with retaining the old idioms as well. They heavily rely on lambda expressions and method references, two new language features that were introduced in Java 8. These are fantastic language features and really helped improve my Java code when I was using Java 8. But I haven't really focused on them in this course simply because it's not a Java language course. It's about Java collections. If you're unfamiliar with that syntax, I'd strongly recommend that you go and have a look at various courses on Pluralsight or books and see more detail about those lambda expressions and method references. Finally, the next module will be on collection factories and further improvements to the collections API that we've seen in recent Java versions.

## **Collection Operations and Factories**

### **Introduction**

Hello, it's Richard again. In this module, we're going to be talking about collection operations and factories. Now the JDK itself ships with a whole series of very useful utilities and a way to creating different collections that don't revolve around particular implementations or interfaces, things like that, and are just general patterns that can be used across different collection types. Those operations and factories are the things we're going to be talking about in this module. So this module contains two big topic areas. Firstly, we're going to be looking at factories. So factories are methods in the JDK that will build you new instances of collections, not just calling constructors on classes like newArrayList or newHashMap, but creating different implementations just for methods that often hide their actual implementation detail that have certain useful properties. The JDK allows us to create unmodifiable, immutable, empty collections. Okay, so different types. And even though unmodifiable and immutable classes may sound very similar, there are some subtle differences. We'll be going through those differences in behaviors in this module and when you might want to use one or the other. Often these things are very, very useful and help you implement good object‑oriented design practices. Then we're going to look at a few operations. So, we have algorithms for finding the max and mean elements within collections, for sorting them, for adding multiple different elements into collections, all sorts of things. These operations will save you a lot of work, having to write them yourself, and they ship with JDK. They're pretty well tested and pretty much bug free at this point.

### **Factory Methods (Live Coding)**

Hello, and welcome to this section of the course. This section of the course is going to be talking about factory methods. I'm going to live code through a few different examples of where we might want to use factory methods and what the different options we have with creating different versions of Java collections are. So firstly, let's take a look at our ShipmentBreaker class here. ShipmentBreaker takes the Shipment class, the same as we saw earlier, adds the products to it, door, window, door, window, floorPanel, and then prepares it and prints out our lightVanProducts. So, if we just run that code, we can very easily see that we've got a couple of glass windows in our lightVanProducts output. Now, the interesting thing is that even though shipment.getLightVanProducts returns this lightVanProducts field, and that's a private field within the Shipment class that we initialize with a sublist, the code that calls getLightVanProducts can do anything it wants with that list. Okay, so we can take the lightVanProducts and we can remove a window from it. And if I print out shipment.getLightVanProducts, I can see that we've modified the underlying list collection here. We've modified the internal state of the Shipment class. And in fact, not only that, but if I take the Shipment class and I print out each of the products within it, we will also see that because that was a view around the products that we have modified the underlying Shipment class. So even though we've added two windows to it originally, we've actually removed one of those windows from within the Shipment class itself, breaking its internal state. Now, object‑oriented programming has this concept called encapsulation, and the idea of encapsulation is that other objects in your system cannot modify or mutate the private state of your classes. And that's what we've done here, we've completely broken encapsulation by calling a getter on a private field and exposing that data. Now, an alternative approach that we might take to solve this problem uses collection factory methods. So, I'm going to take the Collections.unmodifiableList method here, and just wrap that around the lightVanProducts, and we'll see what happens here to our code. Well, we rerun the code, and this time we get an UnsupportedOperationException. That is to say, we have banned the ability to remove values from the lightVanProducts list. That's really useful. This is what we call an unmodifiable collection. You can't add things to it, you can't remove it, you can't clear it, you can't mutate the internal state whatsoever. You can just read data out of it, and that means that we can keep control of the mutation within the Shipment class and not expose the internal state to the outside world. We have achieved a sort of form of encapsulation here and we've reduced the scope for bugs.

### **Factory Method Options (Live Coding)**

Now the unmodifiableList example that we saw in the previous section is one way that we can use a factory method to produce a certain type of invariant of a new collection that we're creating. But actually, there are a few other versions. So, let's take a look at our example here. I'm going to use maps this time. We've got a map from the country's name to its population. So the USA has 328 million people, and the UK has 67 million, and I'm just going to print that map out. Now, let's use that same unmodifiableMap collection that we saw earlier. Well, we saw the unmodifiableList version, but this time we're going to use the unmodifiableMap version. And we're going to create what's known as our unmodifiableCountryToPop. We're just going to put that on the next line so you can read it on the screen. Now the interesting thing here is we have the same exact property as before. So, suppose we put Germany into this map with its 83 million people and we run that code, we'll see that we get the same UnsupportedOperationException. So, this is the unmodifiableMap version of the collection. You cannot modify that unmodifiableCountryToPop variable directly. But, it is interesting. Okay, so if we comment that out, and we show you the unmodifiableCountryToPop value, and I'm going to print it out again afterwards, unmodifiableCountryToPop. And, if in the middle we modify the backing map, the mutableCountryToPop, we can see a more interesting story here. That is to say that our modification of that backing map, mutableCountryToPopulation, has been reflected in the unmodifiableCountryToPop view just like we saw in the views earlier like subList or the keySet and the entrySet for maps themselves. So that means that anything that holds this mutableCountryToPopulation variable like, for example, it could be a private field in the Business class, that controls the modification. And the view, the unmodifiableMap view, allows you to read the changes, but not write the changes, so again, reducing the scope for bugs just like we saw with the list version. UnmodifiableMap is not the only way we can do this kind of thing. So, let's take the Java 10 Map.copyOf(mutableCountryToPopulation) factory method. And we can see there with our copiedCountryToPop variable that I'm going to print it out again beforehand, and I'm going to print it out again after. And this is going to be quite interesting as well. So we'll see that our copied map is not mutable itself. And unlike the unmodifiableCountryToPop, it doesn't reflect the changes in that backing map. It doesn't have a backing map. It's just a copy. We could just use the new HashMap constructor in order to create a copy of a map as well. We'd know is a HashMap, then, which we could modify. So what's the difference here with the Map.copyOf? Well, Map.copyOf for our copiedCountryToPop will produce an unmodifiable copy here. So, if we try and put Germany into the copy, we get an UnsupportedOperationException just like we would with the unmodifiable version of the map. So, this is similar, but changes to the backing collection, they aren't reflected here because it's a copy, not an unmodifiable view. Finally, we're going to look at a slightly different concept. That different concept is of a utility method, our factory method for maps. So what we've got here with this creation of a new HashMap, put in the USA, put in the UK, and eventually put in Germany's map is a lot of code. This is four lines of code just to initialize a variable. So what we could actually do is create a different variable here, which I'm just going to call countryToPopulation. And we could use what was introduced in Java 9, which was a Map.of factory method. So, that would take the UK and its 67 million people. It would take the USA, it's 328 million people, and that's it. We're done. We can just print this out straight off the bat, much quicker, much simpler, much less verbose. We get the same values in that map, the USA and the UK, with the population figures, but we've saved us a lot of code in producing it. It's also worth noting that this Map.of method creates an unmodifiable map. So again, if we try and put Germany and its people into this map, we will get our UnsupportedOperationException. There we go. In this demo, you've seen three different things. You've seen an unmodifiable view, which reflects the changes of its backing collection. You've seen an unmodifiable copy that is unmodifiable itself, and it's just a copy of those values. And finally, you've seen a static factory method for collection here, the Map.of method, that can be used in order to create an immutable collection, and it just takes some initialization values in its parameters.

### **Factory Methods**

So we've seen how the collection factory methods can be used within a live coding example. Let's go through a catalog of all the different options so you can see what's available to you. Empty collections, these are factories from the collections class that will produce us empty lists, maps, and sets. I've included the set examples on these pages as well, even though we haven't covered sets yet, just to show you what they would look like. We'll see sets in more detail in the final module. Empty collections are, as they say, they're collections that can hold zero values. They are, in fact, immutable as well, so you can't add any elements to them. One of the things that's really useful about empty maps or empty lists over calling, say, new ArrayList, which is empty by default, is their reduced memory consumption. These factory methods just refer to static instances and thus don't allocate any objects whatsoever. When you want to pass a value to a method that takes a collection but you don't have any values that you want to pass in yourself, that's a good use case for an empty collection. They can also be used in lots of places where, for example, you need to return a collection over a method, but actually you've got no useful values to return, and it's more efficient to return an empty list than create a new ArrayList. Singletons are collections that only contain one element. These are all factory methods on the collections class. Singleton produces a set, singletonList, a list, and singletonMap, a map. The map takes two arguments, one for the key, one for the value, and the other methods all just take one argument, which is the one value. These collections are immutable, so you can't add any more elements to your singleton. They are more memory efficient than calling the new ArrayList class or the new HashMap and then adding one element in. By default, things on that list get sized to 10 elements once you've added one in. So there's a bunch of wasted memory space there on things like ArrayList when you're only trying to hold one element. They're useful for use cases where you've got a collection that you're either returning or passing to a parameter. But in the case of your specific instantiation of that method, you only actually want to pass one value in. Now let's take a look at the more interesting set of collection factories that got added in Java 9. Interfaces such as list, map, and set have these of static factory methods added to them. I could produce a list of different countries by just saying List.of , UK and USA. This is a massive improvement in boilerplate reduction over previous efforts to, for example, call the new ArrayList constructor and then add multiple elements onto them. And in fact, when they were added, they were added as an alternative to collection literals. So Java has an array literal syntax where we can create some specific instantiation of array, but it was actually decided that these factory methods were so short that the JDK developers didn't need to add any special syntax for instantiating collection literals. These classes are all runtime immutable, so they all implement the list or map or set interface. But when you call the add method or the remove method, they will throw an exception. But those methods is still there, so your code will compile with you having the option of calling those methods to mutate the state, even though they'll never work. These factory methods have multiple different overloads. By default, if you have a varargs argument in Java, there's a bit of overhead with that varargs argument. There's effectively an array that gets created under the hood, passed over a method call boundary, and then thrown away immediately afterwards. So List.of with more than 10 elements does use a varargs parameter just for flexibility, but there have been overloads added for the 1 to 10 argument cases with just multiple numbers of parameters. Map has multiple different ways of instantiating it. Map.of can be used for different specific instantiations up to 10 elements. So here, for example, we create a map of country populations from strings to integer by doing Map.of UK 67 and USA 328 to signify the population in millions at the time of this course's creation. Now the problem with map is that if you want to have more than 10 elements, you want to have that varargs variant of map, you need to have a way of creating multiple varargs, one for the key, one for the values. So a general varargs factory method wouldn't work. This is why the Map.ofEntries method exists. So it takes a varargs of the Map.entry interface. So that's the same type as you can see on the entry set of the map. And in fact, a factory method for Map.entry has been created, as well in Java 9, in order to allow you to build implementations for those Map.entry interface elements. In Java 10, a slightly different variation of this concept of the immutable factory method was added, and that is the immutable copy. Take, for example, what we've done here. We've got a collection of strings, which is our countries. We've added the UK to that collection, we've added the USA, and it's an ArrayList that is backing that implementation so we can mutate that countries variable. We could add elements, we could remove elements, we clear it. Now when we do list.copy of that country's variable, we create new list, and that list is a copy with the same values in as the original ArrayList, but it's an immutable copy. The collection that you copy it into cannot be modified in any way. If you want a mutable copy, most of the collection classes have constructor overloads that take collections as parameters, and those will copy the other elements in. You can do the same thing with maps as well. You can do Map.copyOf and pass in a HashMap implementation. This has the exact same criteria here. It's an immutable copy. If you modify populations, you will not be changing immutable populations. And finally, our unmodifiable views. This seems very, very similar at first to the immutable copies that we saw in the previous slide. We've got our countries instantiated with an ArrayList, we've added the UK, we've added the USA. Now what are we going to do? Well, firstly, Collections.unmodifiableList comes on a different class here. It's on the collections class, not the list interface. And secondly, what we've got here is an unmodifiable view. That is to say, countriesView, that variable, if you try and add anything, remove anything, clear it, no changes will be made within that view. But if you change the countries variable, for example, suppose we added France, that will be visible within anyone reading out of the countriesView. Unmodifiable views are super useful for cases where we want to keep mutation within a class. So we could have a private field called countries that we update over time, and we only expose the countriesView to other classes in your application, and they can read the data but not modify it. You can have an unmodifiableMap, so we've taken our populations example here in the HashMap. If you added a new country into that populations map, you'd be able to read it out of the populationsView but not modify the populationsView.

### **Collection Operations**

In the previous couple of sections, we saw some of the factory methods coming from the Collections class. But actually, that class has a few other operations and algorithms that are defined on it, and that's what we're going to look at in this section. What I've done here is I've created a list of the products, door, floorPanel, and window, using the List.of factory method, and we got our products there using the Java 10 var feature. What do we want to do with this code? Well, let's try and look at some of the operations that we have on the Collections class. Collections has a max and also a min conveniently defined. If we try and put Collections.min, immediately we'll see we get a compile error, and that will tell us that the argument needs to be comparable if we're going to call the Collections.min class. But there's another choice here. We can use the comparator implementation here, as we've seen earlier in this course, and we'll get the minimum product by weight, so we can call that the lightestProduct. And we can do the same thing with max as well. So take the products and call the Product.BY\_WEIGHT, and we'll call that our heavyProduct. Then I'm going to print out the lightestProduct and the heavyProduct and run this code. And we should see that our lightestProduct is the Window, and our heaviest product is the Wooden Door. Fantastic. Very useful algorithm there. Find the max or min value within a given collection based upon a comparator that defines sort order. Let's consider another choice that we have here. At the beginning, I used the List.of factory method, and as we saw earlier, that List.of factory method is very syntactically short, but it does also have one other factor. It produces an immutable list. How would we get a mutable list? We would use the new ArrayList constructor and specify a list of products. But we've got a bit of a problem now. We won't have a convenient way of taking those products and adding these elements in. You know, by default, we've seen the add method that would let us write door, floorPanel, and window. Again, multiple lines of code just to do something very simple here. And actually, there is a better way. Collections has a convenient addAll method that takes a collection and a var args argument of the elements. So this would let us produce a collection with the door, floorPanel, window in, and this collection is mutable as well, so we can still modify it later on in our application if we want to. Now the fun doesn't stop there with the Collections class. There's a number of other operations you have. So, for example, you can shuffle it, which re‑sorts those products according to a random order. There we go. Different order than the one we had before. And there are also other values as well, like Collections has a nice fill method that I found useful before, where it just takes one object and fills all the elements within that list with that one object. So we could take our products, and we can pass in null, and we can just null out the values within the list as well. The point here is really that if there's a certain algorithm that you're thinking of, that it's always worth taking a look at the Collections class first before going off and writing it on your own. It's a real time‑saver, and again, these are tested, shipped implementations, these algorithms that come with the JDK. They'll often be more performant and more reliable than code you'll write yourself, as well as save you time in terms of not having to write it to begin with. In the previous couple of sections, we saw some of the factory methods coming from the Collections class. So, let's wrap up and summarize this module. Collections aren't just about data structures. We've seen different implementations of lists and maps in this course. We've also previously seen the Streams API, which is more about algorithms and data structures. Here we saw some algorithms that were just about modifying collections and creating collections. These are common operations that ship with the JDK. You don't need to implement these in your own application. This saves you time, it lets you focus on business logic, and it reduces the number of bugs that you need to care about. But also, really importantly, the immutable and unmodifiable collections that ship with the JDK reduce the scope for bugs. Mutability of state means that one part of your application can modify state that another part of your application depends on. There's a cost to creating immutable values and handing them about, and there's a cost to those things like List.copyOf. But there is also a lot of value there in terms of reducing the ability for certain parts of your application to introduce bugs into other parts of your application. Similarly, unmodifiable collections provide a halfway house between full immutability and full mutability. You can hand out a read‑only view that other classes can't write to, but it can still reflect rights on another value that you hold yourself.

## **Collection with Uniqueness: Sets**

### **Introduction**

Hello and welcome to the final module of this course called Sets, with Me, Richard Warburton. Sets are collections that have uniqueness as a fundamental property. So Java has a concept of an equals method. If calling equals on two objects returns true, you can say that they're the same object. Sets are collections that only have distinct elements, so there are no two objects within a set in which if you call equals on both of them, it will return true. Implementations of sets may require you to implement other methods within your object such as hashCode, or provide a defined sort order in order to efficiently implement this equality contract. In this course, we're going to be talking about a few different things. Firstly, we're going to do a live coding demo about the why and how of sets themselves, so explaining why you might want to use a set and how they work and what the kind of features are that are available there. Then there's the hashCode and equals method. In practice, HashSet is the most commonly used set implementation, and in order to use it correctly, just like with HashMap, you need to have your hashCode and equals contract correct. So we're going to go through that contract and show you what might happen if you get it wrong. Finally, as with all these different collections, we're going to be looking at the performance tradeoffs and the features that distinguish between different set implementations. Here, we're going to have a look a HashSet and TreeSet, which the two most commonly used set implementations in the JDK.

### **Set Features (Live Coding)**

So let's take a look at an example of when we might actually want to use a set. Here is our ProductCatalogue. Our ProductCatalogue is going to have the ability to add products in from suppliers as we accept those suppliers into our business, and then it will let us iterate over the products that we have within the ProductCatalogue. Here is our Supplier class. Each supplier has a list of products and a name. Our ProductCatalogue has a test. The test takes a door, floorPanel, window, and two suppliers. Firstly, there's John's Glazing, who just do windows, and secondly, there's All Purpose Supplies Ltd. who supply us with doors, floor panels, and windows. So what we do is we add both those suppliers in and check that we've got a door, a floor panel, and a window as products that we can sell. Now, if we run this we immediately see we have a NullPointerException, and that is because our ProductCatalogue hasn't been implemented. So let's first of all try and implement this product with a list. So, we're going to create a list of products, and back it with an ArrayList, and then we're going to addAll of the products from our supplier to that list, and then we are going to return the iterator of that products list. If we flip back to our test, we can run this test, and unfortunately, we'll see that our test fails. The reason is that it says here you have no match for your glass window product. And what actually that means is that you were expecting your ProductCatalogue to just contain a door, a floor panel, and a window, but in practice, it contains two different windows. Even though these are duplicate products, you've got two entries in your list because you've added them twice, once from John's Glazing and once from All Purpose Supplies. So, let's go and take our code and refactor it a little bit. So what we could do is instead of adding all the products, so let's refactor this code now in order to try and implement it a different way. What we could do is we could say, get the products from our supplier and call that supplierProducts, and then take the supplierProducts and for each product within that list we could say, if our existing products list does not contain the product, then we will add in that product. And if we rerun the test, we find the test passes. But this is a super inefficient way of implementing this code. It's inefficient in terms of your time because you have to change just a simple addAll to be this complicated loop with a bit of logic in that you have to check for a product, and it's also inefficient in terms of time because your array list, every time you add a product it has to scan through the entire existing list of products in order to ensure that there's no duplicates. And this is where sets come in. So, if we just took a set of products, and it's a HashSet, I won't really be able to demonstrate how it's faster to you, but I will demonstrate how the code is simpler. We can just go back to our nice addAll method here, and if we rerun the test now, we can see the test passes, which is fantastic. And that's because the addAll method within a set will delegate to the add method, and the add method will only add in products once it doesn't have any duplicates in the set. And set implementations are written in such a way that they are efficient and you only get one element in there in a much, much faster way than having to scan through the entire collection of values every time around.

### **Hashcode and Equals**

So let's talk a little bit about the hashCode and equals contract that we have within Java. The hashCode and equals contract defines a kind of fine print to an API. A lot of APIs have a sort of contract within them. The contracts are assumptions that are being made by the HashMap or HashSet class within the JDK and there are obligations on your code that you need to meet in order for HashMap to work. If you don't meet the hashCode equals contract within your code, you can find situations where you get duplicate objects within your set, which breaks the whole invariant and purpose of using sets to begin with. The reason behind this is that the HashMap and HashSet classes use the hashCode method to find a slot and then they only check the equals methods between either the value in the HashSet case or the key in the HashMap case using the equals method once you've hit that slot. If two different objects that are equal result in different hashCodes, then they can end up in different slots and that means that the contract is very simple. If you're object equals another object, then the hashCode of that object should also equal the hashCode of the other object. Note, this is a one‑way implication. Two different objects can have the same hashCode, but if they're the same object, they must have the same hashCode. There are two different kinds of equality, either reference‑based equality or value‑based equality. So reference‑based equality is when two objects are equal because they're the same exact object and value‑based equality is when they're equal because they have the same values in their fields. So, for example, a person's name and weight and ID might be the same. You don't need to use all the fields for your value based equality. Often if you have a specific ID field, you just use that. Now what the implication of this is is if you have a reference‑based equality system between objects, you just inherit the equals method from Java line object, but if you've got a value‑based equality system, you need to have a custom equals method and that is where you need to get the equals hashCode contract from. This Java line object correctly implements the hashCode equals contract itself. The first thing you need to think about when creating a hashCode is that you want to combine the hashCode information from each field, so a standard approach is to have a variable called result, you're going to initialize result to a number that will be a prime number, and definitely not 0, 17 is often a good choice. That result at the end of your hashCode method is going to be returned and that will be the new hashCode value. If you've got a normal object field, you can call the hashCode method on that field and add it into the result previously. It's important not just to add the values in, because this results in a poor distribution of hashCode values and often can result in poor HashMap or HashSet performance. So we multiply the previous result value here by prime, it's 31 in the example on screen, but it could really be any prime. If you just call the hashCode method on an array, you will get a reference‑based hashCode just like we saw at the beginning of this course. If you call toString or equals on arrays, you get useless values. Well, at least you don't get value‑based values. Here, it's the same thing. We want to call Arrays.hashCode in order to get the correct hashCode method for an array. Now, when it comes to primitive values like ints or longs, as of Java 9 or later, all the primitive relevant classes, like Long with a Capital L or Big I Integer have had a hashCode static method added to them. You just pass the value of your field into the hashCode method, and you get an appropriate hashCode value. So if you're using Java 8 or later, you should use these convenience methods. Unfortunately, if you are stuck before Java 8 or wondering why on earth code that was written before Java 8 reads with some weird incantation. Older JDK versions need different approaches. So, for example, if you're going to use a long value, L in this example, it is best to bit shift it to get the 32 bits of information because you're hashCode is a 32‑bit integer and then cast it to an int value at the end. If you're using something like a float or a double, calling the float to int bits method will get you an int value that's relevant for a float. It's also worth noting that if you're using a modern ID, which you definitely should be doing, that can auto generate implementation of your hashCode and equals methods for you based upon just saying, here are some fields, go do it for me. In general, if you are using the ID, and as I say, you should be, just use the auto generate method, it will save you time and it will result in less bugs. But it's important that you still understand the rules in order that you know what the correct code is here if you're reviewing someone else's code or if you're reading other code that you might see in a legacy system. As of Java 7 or later, a method was added called objects.hash that just takes a var args array of fields and that can be used to generate a correct hashCode from those fields. Now, this has a performance overhead to it because there is a var args array creation, so a lot of people prefer not to use it, but it is a convenience method that's there and will result in less code. The final point, and I think this is probably the error that most people make when they're implementing hashCode methods, or at least the bugs that I've seen most frequently, is that you always want to use the same fields when calculating your hashCode value as you do an equals. Often, errors in the hashCode equals contract aren't written when classes are first written, they're introduced into classes over time. You'll add a field to the class, you then realize you need it in your equals method, and you forget to update the hashCode method. Often, if I was adding a new field to a class that I wanted to include in my equals or hashCode method, I would just delete the equals and hashCode method at the field and then get the ID to regenerate it for me. It's often easier to do it that way.

### **Set Implementations (Live Coding)**

Okay, and finally, let's wrap things up by talking about the different implementations you have that are available for general purpose sets. Firstly, HashSet. HashSet is based upon HashMap and actually it uses the HashMap implementation in order to implement the HashSet. You can think about a HashSet as being a HashMap where there is only one value, and whenever you call an operation like is an element within your set, it's just delegating to the contains key method. Whenever you call an operation like contains, it just delegates to contains key, for example, so similarly to HashMap, it uses the hashCode method and it looks up a slot within a backing array, checks the entries, the entries can become linked lists if there are collisions, the equals method is used to check whether the value is unique compared to those other values. Very, very simple, and in fact, for any given map implementation, you can create a wrapping set just by using the key set capability of the map to implement your set. HashSet is a very good general purpose implementation, I would say 90% of the time you want a set, just use a HashSet out of the box, it's very fast, in most use cases in practice, it will be faster than TreeSet due to the reduced amount of pointer interaction and pointer chasing with HashMap and TreeMap. Then there is TreeSet. So TreeSet is based upon TreeMap. You're using a red/black binary tree with a defined sort order. That means as the tree starts to contain more elements, the height of the tree expands bounded by log to the base 2 of n where n is number elements, and that is how the asymptotic performance of different operations on TreeSet like adding, removal, contains, or perform. Just like TreeMap, it provides extra features over HashSet. That is to say it implements the SortedSet and NavigableSet interfaces. These mean that the entries within your TreeSet have a defined order so you can have capabilities a bit like a list where you had a defined order within the elements, but also with the uniqueness capabilities of the TreeSet. And it also means that when you're using TreeSet, you have certain extra requirements, that is to say, you either need your elements to implement comparable or you need to provide a comparator to the constructor of the TreeSet in order to use it. So I'm just going to do a simple example here of the useful features that a TreeSet has to offer us. I've got a WeightAwareProductCatalogue here and the test from my WeightAwareProductCatalogue is it's got this new method in called findLighterProducts, you pass in the product, and it gives you products that have lower weights than it. So I say a door and it gives me a window and a floor panel. The initial implementation of my WeightAwareProductCatalog here is basically the same as my product catalog from the after version of the previous live coding section. So it stores all those products in a HashSet, it returns null for this FindLighterProducts method, and it has an iterator for the products. And if we run this test, we can see that it fails. And in fact our value here is null because we haven't implemented the method yet. Also note that I've changed from a contains in any order matcher here to a contains method so they're in a specified order. So I'm going to change around my set here, and instead of having a set interface, I'm going to have a NavigableSet interface and the NavigableSet interface has a whole series of methods that let us take elements and defined other elements like a ceiling or a floor or higher elements, lower elements, things like that take advantage of the fact that we can go through elements in order and I'm going to initialize that with a TreeSet, and a TreeSet, if we look at the constructors for that TreeSet, you either need to provide another sorted set or a comparator or the type has to be comparable itself. So what I want to provide here is the Product.BY\_WEIGHT comparator, so that good old familiar favorite that we've been using throughout this course that sorts products by weight. So when I enter products into my TreeSet, they will be added by their weight. And when we come to find lighter products, I can just use that products map and provide the headset method so that will find anything in the TreeSet that is before that element and we can use the overload that takes a Boolean to define whether it's inclusive or not. So I can say, take that product and I'm going to say inclusive false because we want to find lighter products, not products with the same weight or lighter. And if I rerun this test, we can see that it passes. Fantastic.

### **Conclusion**

Okay, let's wrap up and conclude both this module and the entire collections course. Firstly, sets are a very commonly used collection, not as commonly used as lists and maps, but a lot more than things like queues. We saw there were different implementations for different purposes, so HashSet was our good general‑purpose implementation, and usually, the fastest. TreeSet was useful when we wanted to have the SortedSet or NavigableSet features and keep our elements in a given encounter order. You should always try and remember to get the hashCode and equals contract correct for objects. That's true in general, but if you're going to use HashSet and HashMap, it's absolutely critically important because if you have those hashCode equals contracts incorrect, then the HashMap and HashSet classes cannot function as you would expect and not as they're described in the documentation. Finally, let's come back to our Roman coin image from the beginning of the course. If you get the fundamentals right, you're on the way to building a powerful piece of software. Collections are a key part of that Java fundamentals and a key part of becoming a good Java programmer. There's lots and lots of useful functionality within the JDK collections framework, stuff that you can use in your application on a day‑to‑day basis. And picking the correct collection can really help you write simpler code, more readable code, less buggy code, and also faster code, compared with inventing your own collections and building your own collection or framework itself. I really strongly hope that having finished this course, and congratulations to you for doing so, that you take the opportunity to see how you can use some of these collections in your own programming projects. I promise you it'll really help you, and I hope you enjoy using the Java collections framework.