**The Collections Framework:**

**Storing Data Using the Collections Framework:**

Introducing the Collections Framework:

The Collections Framework is the most widely used API of the JDK. Whatever the application you are working on is, odds are that you will need to store and process data in memory at some point.

The history of data structures goes back nearly as far back as computing itself. The Collections Framework is an implementation of the concepts on how to store, organize, and access data in memory that were developed long before the invention of Java. The Collections Framework does this in a very efficient way, as you are going to see.

The Collections Framework was first introduced in Java SE 2, in 1998 and was rewritten twice since then:

* in Java SE 5 when generics were added;
* in Java 8 when lambda expressions were introduced, along with default methods in interfaces.

These two are the most important updates of the Collections Framework that have been made so far. But in fact, almost every version of the JDK has its set of changes to the Collections Framework.

You will learn in this part are the most useful data structures the Collections Framework has to offer, along with the patterns you will be using to manipulate this data in your application.

The first thing you need to know is that, from a technical point of view, the Collections Framework is a set of interfaces that models different way of storing data in different types of containers. Then the Framework provides at least one implementation for each interface. Knowing these implementations is as important as the interfaces, and choosing the right one depends on what you need to do with it.

Finding Your Way in the Collections Framework:

The amount of interfaces and classes in the Collections Framework may be overwhelming at first. Indeed, there are many structures available, both classes and interfaces. Some have self-explanatory names, like LinkedList, some are carrying **behavior**, like **ConcurrentHashMap**, some may sound weird, like **ConcurrentSkipListMap**.

You are going to use some of these elements much more often than others. If you are already familiar with the Java language, you probably came across **List**, **ArrayList** and **Map**

This topic focuses on the most widely used structures of the Collections Framework.

That being said, you need to have the big picture of what is there for you in the Collections Framework.

First, the framework consists of interfaces and implementations. Choosing the right interface means you need to know what functions you want to bring to your application. Is what you need consists in:

* storing objects and iterating over them?
* pushing your object to a queue and popping them?
* retrieving them with the use of a key?
* accessing them by their index?
* sorting them?
* preventing the presence of duplicates, or null values?

Choosing the right implementation means you need to know how you are going to use these functionalities:

* Will accessing your objects be done by iterating, or random, indexed access?
* Will the objects be fixed at the launch of your application, and will not change much over its lifetime?
* Will the amount of objects be important with a lot of checking for the presence of certain objects?
* Is the structure you need to store your objects in will be accessed concurrently?

The Collections Framework can give you the right solution to all of these problems.

There are two main categories of interfaces in the Collections Framework: **collections** and **maps.**

Collections are about storing objects and iterating over them. The **Collection** interface is the root interface of this category. In fact, the **Collection** interface extends the **Iterable** interface, but this interface is not part of the Collections Framework.

A Map stores an object along with a key, which represents that object, just as a primary key represents an object in a database, if you are familiar with this concept. Sometimes you will hear that maps store key/value pairs, which exactly describes what a map does. The **Map** interface is the root interface of this category.

There is no direct relationship between the interfaces of the **Collection** hierarchy and the **Map** hierarchy.

Along with these collections and maps, you also need to know that you can find interfaces to model queues and stacks in the **Collection** hierarchy. Queues and stacks are not really about iterating over collections of objects, but since they have been added to the Collection hierarchy, it turns out you can do that with them.

There is one last hierarchy that you also need to know, which is the **Iterator** hierarchy. An iterator is an object that can iterate over a collection of objects, and it is part of the Collections Framework.

That makes two main categories, **Collection** and **Map**, a subcategory, **Queue**, and a side category, **Iterator**.

Avoiding Using Old Interfaces and Implementations:

The Collections Framework was only introduced in Java 2, meaning that there was a life before it. This life consisted in several classes and interfaces that are still in the JDK, to preserve backward compatibility, but that you should not use anymore in your applications.

Those classes and interfaces are the following:

**Vector** and **Stack**. The Vector class has been retrofitted to implement the **List** interface. If you are using a vector in a non-concurrent environment, then you can safely replace it with **ArrayList**. The **Stack** class extends **Vector** and should be replaced by **ArrayDeque** in non-concurrent environments.

The **Vector** class uses the **Enumeration** interface to model its **iterator**. This interface should not be used anymore: the preferred interface is now the **Iterator** interface.

**HashTable**: This class has been retrofitted to implement the **Map** interface. If you are using instances of this class in a non-concurrent environment, then you can safely replace its use with **HashMap**. In a concurrent environment, **ConcurrentHashMap** can be used as a replacement.

Why Choose a Collection Over an Array?

You may be wondering why you should bother learn the Collection Frameworks when you may have the feeling that putting your data in a good old array does the job.

The fact is, in any case, if you have a solution that is simple, that you master well, and that fits your needs, then you should definitely stick with it!

What can a collection do for you, that an array cannot?

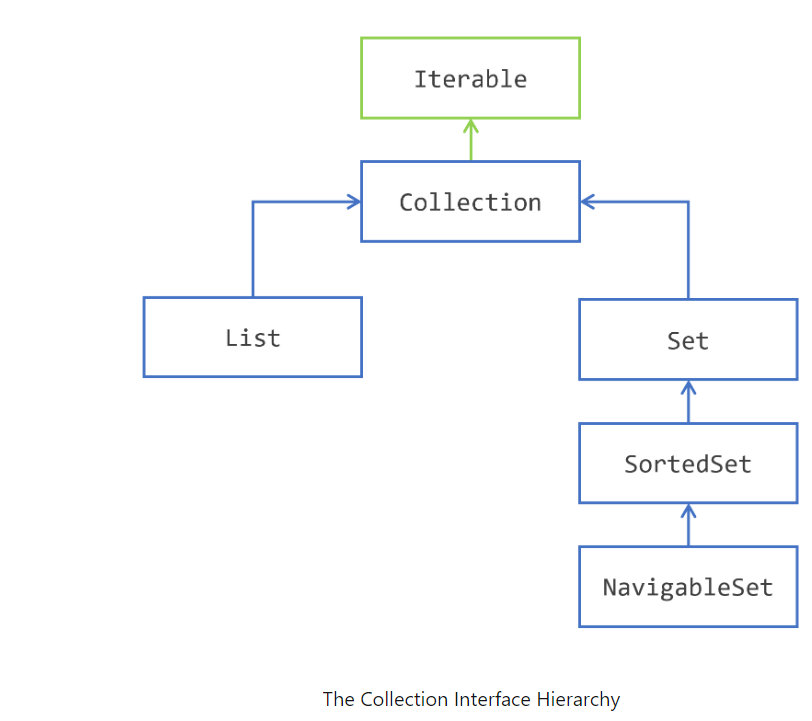
* A collection tracks the number of elements it contains
* The capacity of a collection is not limited: you can add (almost) any amount of elements in a collection
* A collection can control what elements you may store in it. For instance, you can prevent null elements to be added
* A collection can be queried for the presence of a given element
* A collection provides operations like intersecting or merging with another collection.

This is just a small sample of what a collection can do for you. In fact, since a collection is an object and given that an object is extensible, you can add any operation you need on most of the collections provided by the JDK. It is not possible to do this with an array, because an array is not an object in Java.

**Getting to Know the Collection Hierarchy:**

Avoiding Getting Lost in the Collection Hierarchy:

The Collections Framework is divided in several hierarchies of interfaces and classes. The first one you need to understand is the following: the Collection interface hierarchy.

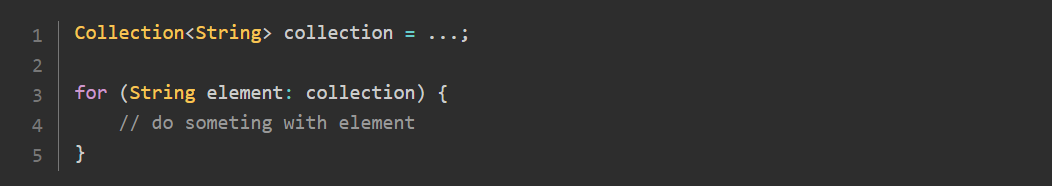


The Iterable Interface:

The first interface of this hierarchy is the **Iterable** interface, and it is in fact not part of the Collections Framework. It is still worth mentioning here because it is the super interface of the **collection** interface, and thus of all the interfaces of this hierarchy.

The **Iterable** interface is an addition of Java SE 5 (2004). An object that implements **Iterable** is an object that you can iterate over. It has been added in Java SE 5 along with the for each pattern of code.

You may be already familiar with this way of iterating over the elements of a **Collection**:



You may already know that you can iterate over any collection using this pattern, or any array. It turns out that in fact any instance of **Iterable** may be used here.

To implement the **Iterable** interface is really easy: all you need to do is provide an instance of another interface, **Iterator**, that you are going to see in the following.

**Storing Elements in a Container with the Collection Interface:**

All the other interfaces are about storing elements in containers.

The two interfaces **List** and Set both share a common behavior, which is modeled by the **Collection** interface. The **Collection** interface models several operations on containers of elements. Without diving in the technical details (yet!), here is what you can do with a **Collection:**

* add or remove elements;
* test for the presence of a given element;
* ask for the number of elements contained, or if this collection is empty;
* clear this content.

Since a Collection is a set of elements, there also set operations defined on the **Collection** interface:

* testing for the inclusion of a set in another set;
* union;
* intersection;
* complement.

Lastly, the **Collection** interface also models different ways of accessing its elements:

* you can iterate over the elements of a collection, through the use of an iterator;
* you can create a stream on these elements, that can be parallel.

Of course, all these operations are also available on **List** and **Set**. So, what does make the difference between a plain instance of **Collection** and an instance of **Set** or an instance of **List**?

**Extending Collection with List:**

The difference between a **List** of elements and a **Collection** of elements, is that a **List** remembers in what order its elements have been added.

The first consequence is that if you **iterate** over the elements of a **list**, the first element you will get is the first that has been added. Then you will get the second one, and so on until all the elements have been seen. So the order you will iterate over the elements is always the same, it is fixed by the order in which these elements have been added. You do not have this guarantee with a plain Collection nor for a **Set.**

It turns out that some implementations of **Set** provided by the **Collections Framework** happen to always iterate over the elements in the same order. This is an accidental effect, and your code should not rely on this behavior.

There is a second consequence, maybe not as clear as the first one, which is that the elements of a **list** have an **index**. **Querying** a collection for its first element does not make sense. Querying a **list** for its first element does make sense, since a **list** does remember that.

How are those indexes handled? Well, once again, this is the responsibility of the implementation. The first role of an interface is to specify a behavior, not to tell how an implementation should achieve that.

As you will see it, the **List** interface adds new operations to the **Collection interface**. As the elements of a list have an index, you can do the following with that index.

* Get an element at a specific index, or delete it
* Insert an element or replace an element at a specific position
* Get a range of elements between two indexes.

**Extending Collection with Set:**

The difference between a **Set** of elements and a **Collection** of elements, is that you cannot have **duplicates** in a **Set**. You can have several instances of the same class that are equal in a Collection, or even the same instance more than once. This is not allowed in a **Set**. How is this enforced is the responsibility of the implementation, you will see that later in this tutorial.

One of the consequences of this behavior, is that adding an element to a **Set** may fail.

Then you may ask yourself: can I have a container that prevents having **duplicates**, and in which elements have an index? The answer is not that simple. The **Collections Framework** gives you an implementation of **Set** with which you will **iterate** over the elements always in the same order, but these elements do not have an index, so this class does not implement **List**.

This difference in behavior does not bring any new operations in the **Set interface.**

**Sorting the element of a Set with SortedSet and NavigableSet:**

The **Set** interface has itself two extensions: **SortedSet** and **NavigableSet**.

The **SortedSet** interface maintains its elements sorted in the ascending order. Once again, how this is enforced is the responsibility of the implementation, as you will see it later.

To be able to sort them, a **SortedSet** needs to compare your elements. How can it achieve that? Well, there are two standard mechanisms defined in the Java language for that.

* Your elements may implement the Comparable interface, and provide a **compareTo()** method
* You give a Comparator to the **SortedSet** so that it can compare them.

Even if your elements are Comparable, you can still provide a Comparator when building a **SortedSet**. This may prove useful if you need to sort your elements in an order that is different from the one implemented in the **compareTo()** method.

What is the difference between sorting and ordering? A **List** keeps its elements in the order they have been added, and a **SortedSet** keeps them sorted. Sorting elements means that the first element you get while traversing a set will be the lowest one, in the sense of a given comparison logic. Ordering elements means that the order in which you added them in a list is kept throughout the life of this list. So the first element you get while traversing a list is the first that has been added to it.

The **SortedSet** adds several operations to Set. Here is what you can do with a **SortedSet**.

* You can get the **lowest** element, and the **largest** element of the **set**
* You can extract a **headSet** and a **tailSet** of all the elements lesser than, or greater than a given element.

Iterating over the elements of a **SortedSet** will be made from the lowest element to the greatest.

The **NavigableSet** does not change the behavior of a **SortedSet**. It adds several vey useful operations on **SortedSet**, among them the possibility to iterate over the elements in the descending order. You will see more details ont that later.

**Storing Elements in a Collection:**

**Exploring the Collection Interface:**

The first interface you need to know is the **Collection interface**. It models a plain collection, which can store elements and gives you different ways to retrieve them.

If you want to run the examples in this part, you need to know how to create a collection. We have not covered the **ArrayList** class yet, we will do that later.

You can run the following example. Here, you create an instance of the **Collection** interface using the **ArrayList** implementation. The generics used tells the Java compiler that you want to store String objects in this collection. **ArrayList** is not the only implementation of **Collection** you may use. More on that later.

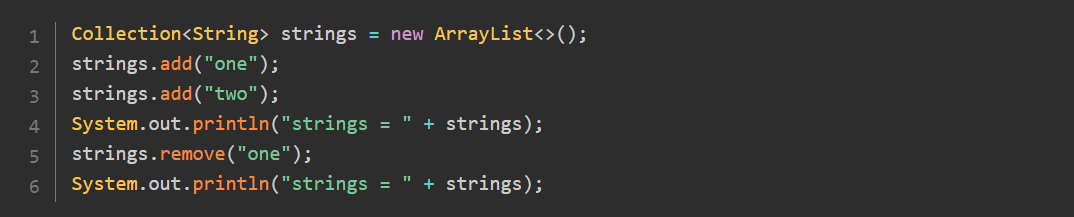
**Methods That Handle Individual Elements:**

Let us begin by storing and removing an element from a collection. The two methods involved are **add()** and **remove().**

**add(element):** adds an element in the collection. This method returns a **boolean** in case the operation failed. You saw in the introduction that it should not fail for a **List**, whereas it may fail for a **Set**, because a set does not allow duplicates.

**remove(element):** removes the given element from the collection. This method also returns a boolean, because the operation may fail. A remove may fail, for instance, when the item requested for removal is not present in the collection

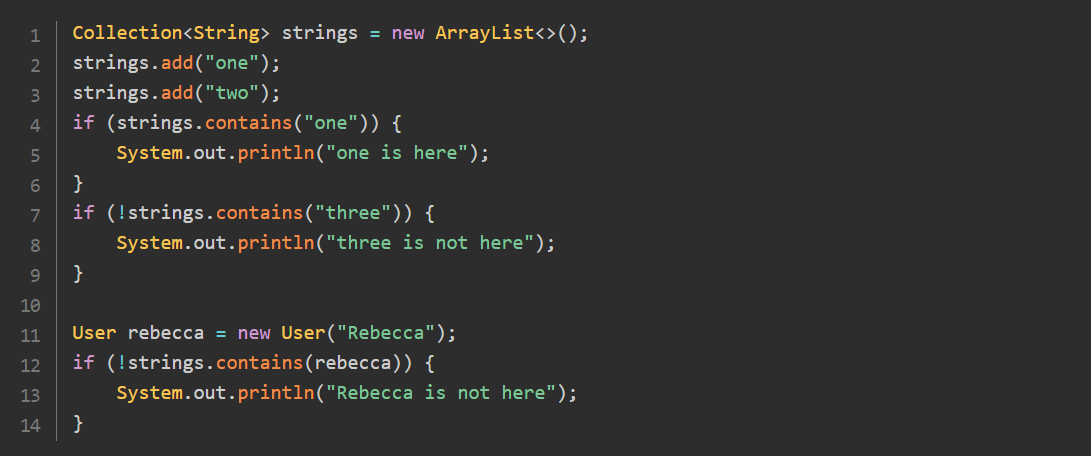
You can run the following example. Here, you create an instance of the **Collection** interface using the **ArrayList** implementation. The generics used tells the Java compiler that you want to store String objects in this collection. **ArrayList** is not the only implementation of **Collection** you may use. More on that later.



Running the previous code should print the following:



You can check for the presence of an element in a collection with the **contains()** method. Note that you can check the presence of any type of element. For instance, it is valid to check for the presence of a **User** object in a collection of **String**. This may seem odd, since there is no chance that this check returns **true**, but it is allowed by the compiler. If you are using an IDE to test this code, your IDE may warn about testing for the presence of a **User** object in a collection of String objects.



Running this code produces the following:



**Methods That Handle Other Collections:**

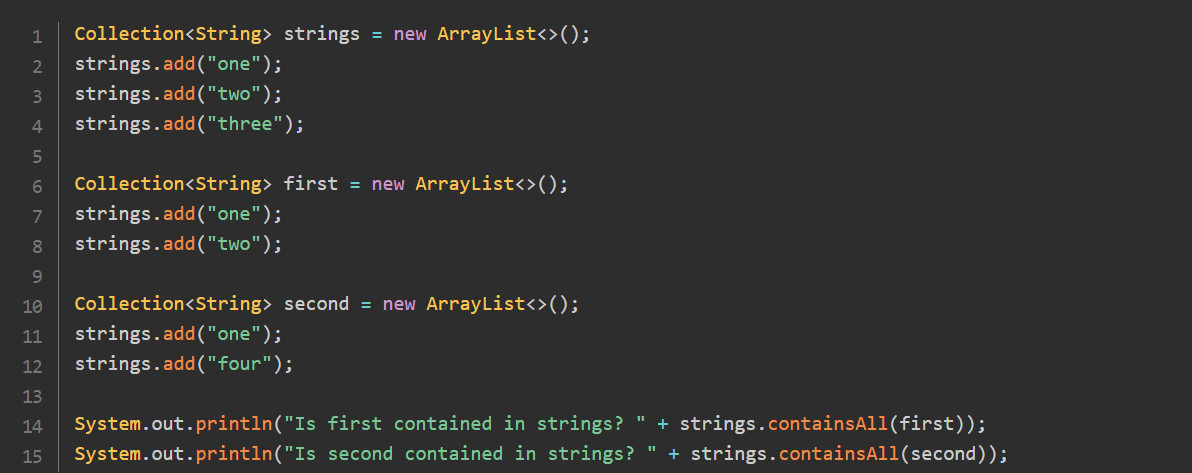
This first set of methods you saw allows you to handle individual elements.

There are four such methods: **containsAll()**, **addAll(),** **removeAll()** and **retainAll().** They define the four fundamental operations on set of objects.

* **containsAll():** defines the inclusion
* **addAll():** defines the union
* **removeAll():** defines the complement
* **retainAll():** defines the intersection.

The first one is really simple: **containsAll()** takes another collection as an argument and returns **true** if all the elements of the other collections are contained in this collection. The collection passed as an argument does not have to be the same type as this collection: it is legal to ask if a collection of **String,** of type **Collection<String>** is contained in a collection of User, of type **Collection<User>**.

Here is an example of the use of this method:

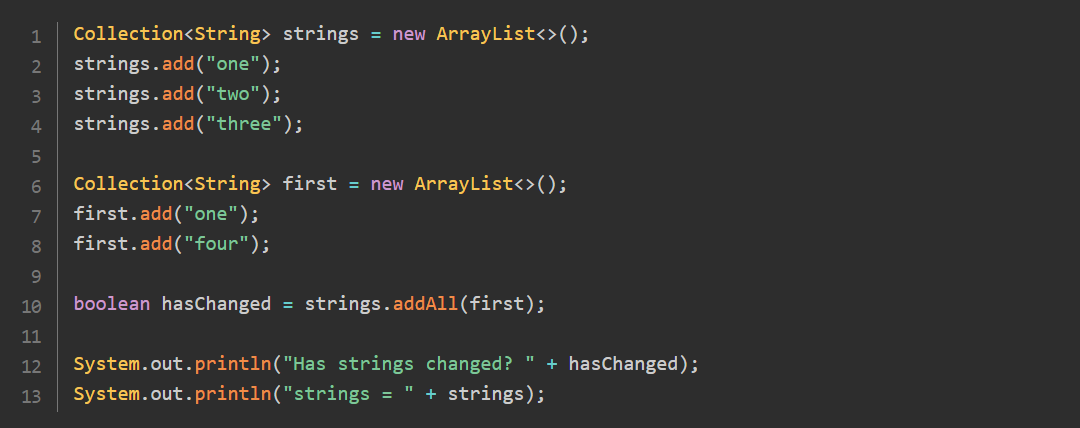


Running this code produces the following:

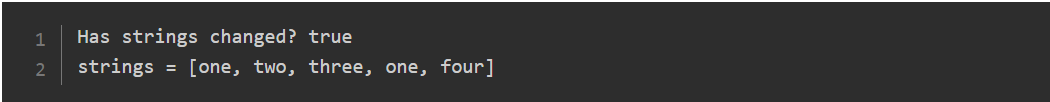


The second one is **addAll().** It allows you to add all the elements of a given collection to this collection. As with the **add()** method, this may fail for some elements in some cases. This method returns **true** if this collection has been modified by this call. This is an important point to understand: getting a **true** value does not mean that all the elements of the other collection have been added; it means that at least one has been added.

You can see **removeAll()** in action on the following example:



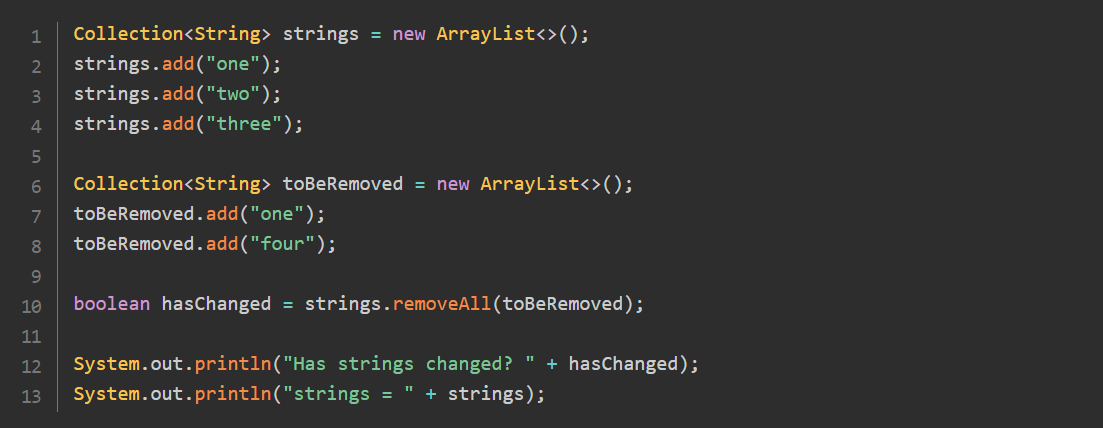
Running this code produces the following result:



You need to be aware that running this code will produce a different result if you change the implementation of **Collection**. This result stands for **ArrayList**, as you will see in the following, it would not be the same for **HashSet**.

The third one is **removeAll().** It removes all the elements of this collection that are contained in the other collection. Just as it is the case for **contains()** or **remove(),** the other collection can be defined on any type; it does not have to be compatible with the one of this collection.

You can see **addAll()** in action on the following example:

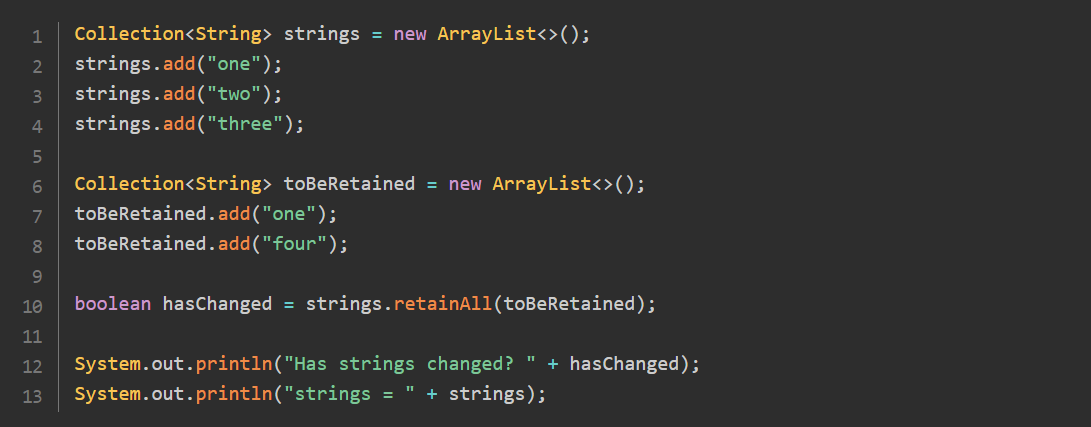


Running this code produces the following result:



The last one is **retainAll().** This operation retains only the elements from this collection that are contained in the other collection; all the others are removed. Once again, as it is the case for **contains()** or **remove(),** the other collection can be defined on any type.

You can see **retainAll()** in action on the following example:



Running this code produces the following result:

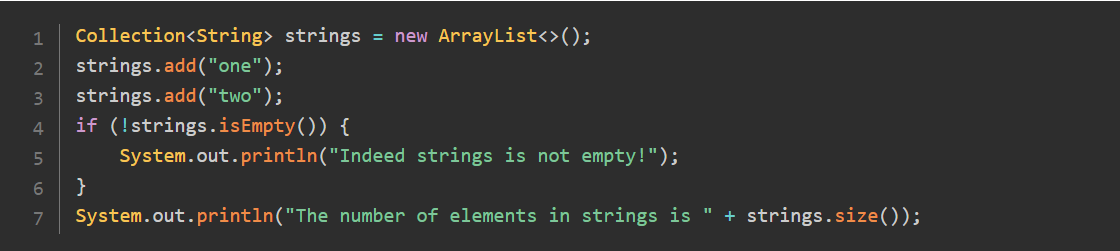


**Methods That Handle The Collection Itself:**

Then the last batch of methods deal with the collection itself.

You have two methods to check the content of a collection.

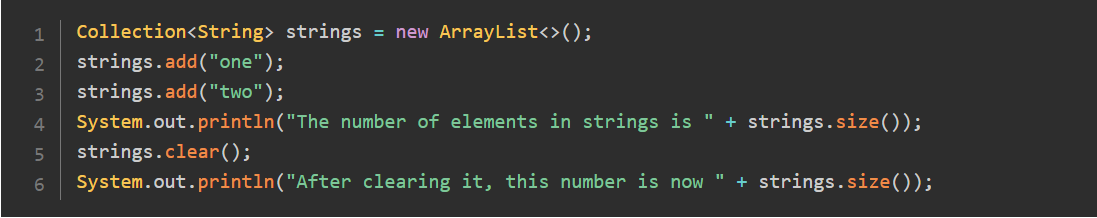
* **size():** Returns the number of elements in a collection, as an int.
* **isEmpty():** Tells you if the given collection is empty or not.



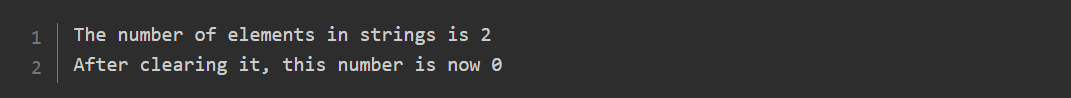
Running this code produces the following:



Then you can delete the content of a collection by simply calling **clear()** on it.



Running this code produces the following:



**Getting an Array of the Elements of a Collection:**

Even if storing your elements in a collection may make more sense in your application than putting them in an array, there are still cases where getting them in an array is something you will need.

The **Collection** interface gives you three patterns to get the elements of a collection in an array, in the form of three overloads of a **toArray()** method.

The first one is a plain **toArray()** call, with no arguments. This returns your elements in an array of plain objects.

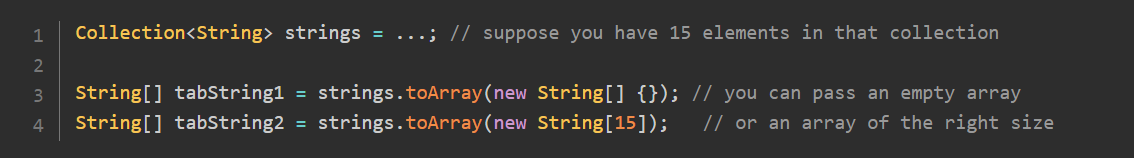
This may not be what you need. If you have a **Collection<String>,** what you could prefer is an array of **String**. You can still cast **Object[]** to **String[],** but there is no guarantee that this cast will not fail at runtime. If you need type safety, then you can call either of the following methods.

**toArray(T[] tab)** returns an array or **T: T[]**

**toArray(IntFunction<T[]> generator),** returns the same type, with a different syntax.

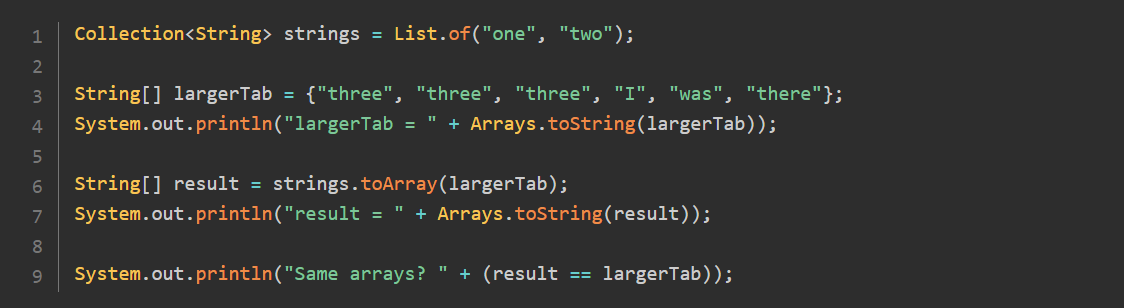
What are the differences between the last two patterns? The first one is readability. Creating an instance of **IntFunction<T[]>** may look weird at first, but writing it with a method reference is really a no brainer.

Here is the first pattern. In this first pattern, you need to pass an array of the corresponding type.

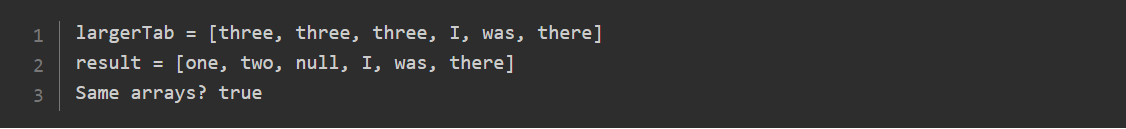


What is the use of this array passed as an argument? If it is big enough to hold all the elements of the collection, then these elements will be copied in the array, and it will be returned. If there is more room in the array than needed, then first of the unused cell of the array will be set to null. If the array you pass is too small, then a new array of the exact right size is created to hold the elements of the collection.

Here is this pattern in action:

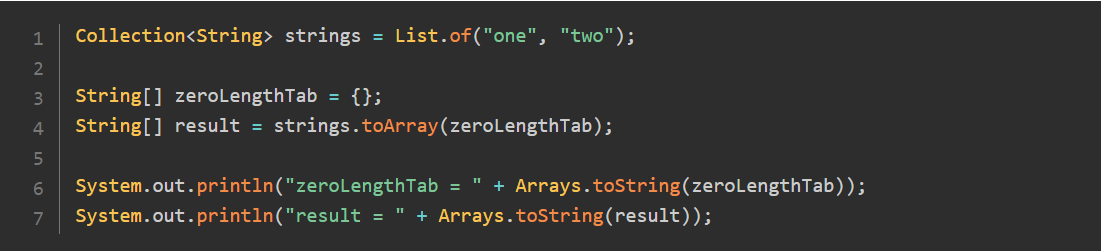


Running the previous code will give you:



You can see that the array was copied in the first cells of the argument array, and null was added right after it, thus leaving the last elements of this array untouched. The returned array is the same array as the one you gave as an argument, with a different content.

Here is a second example, with a zero-length array:

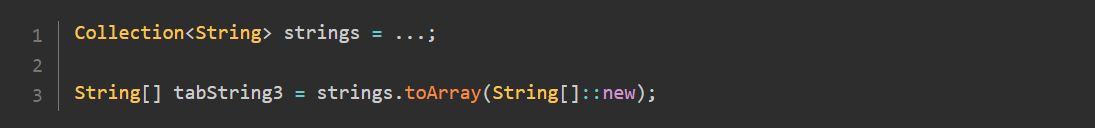


Running this code gives you the following result:



A new array has been created in this case.

The second pattern is written using a constructor method reference to implement **IntFunction<T[]>:**



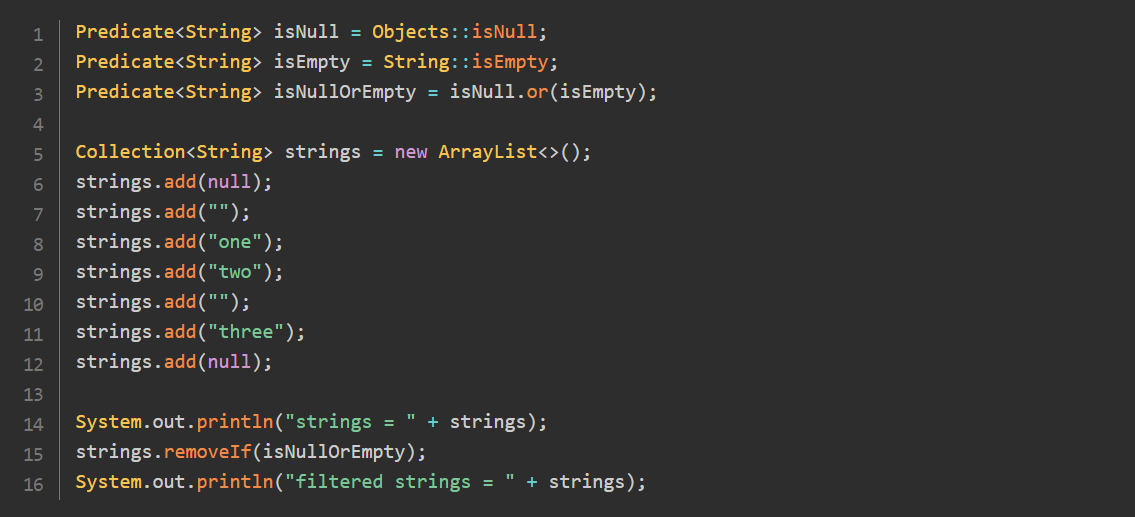
In that case, a zero-length array of the right type is created with this function, and this method then calls to **toArray()** with this array passed as an argument.

This pattern of code was added in JDK 8 to improve the readability of the **toArray()** calls.

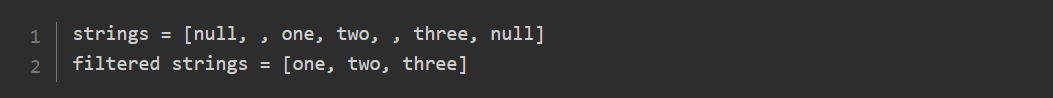
**Filtering out Elements of a Collection with a Predicate:**

Java SE 8 added a new feature the **Collection** interface: the possibility to filter out elements of a collection with a predicate.

Suppose you have a **List<String>** and you need to remove all the null strings, the empty strings and the strings longer than 5 characters. In Java SE 7 and earlier, you can use the **Iterator.remove()** method to do that, calling it in an **if** statement. You will see this pattern along with the **Iterator** interface. With **removeIf(),** your code becomes much simpler:



Running this code produces the following result:



Once again, using this method will greatly improve the readability and expressiveness of your application code.

**Choosing an Implementation for the Collection Interface:**

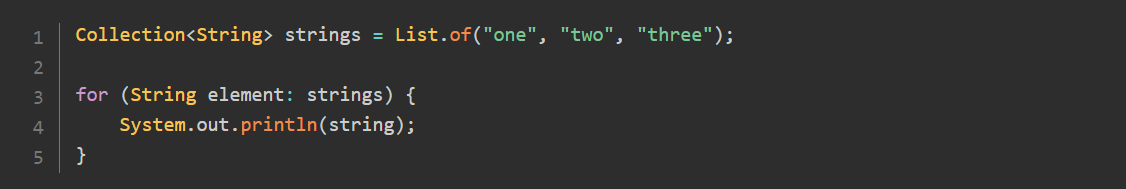
In all these examples, we used **ArrayList** to implement the **Collection** interface.

The fact is: the Collections Framework does not provide a direct implementation of the Collection interface. **ArrayList** implements **List**, and because **List** extends **Collection**, it also implements **Collection**.

If you decide to use the **Collection** interface to model the collections in your application, then choosing **ArrayList** as you default implementation is your best choice, most of the time. You will see more discussions on the right implementation to choose in later in this tutorial.

**Iterating over the Elements of a Collection:**

**Your simplest choice to iterate over the elements of a collection is to use the for-each pattern.**



Running this code produces the following result:



This pattern is very efficient, as long as you only need to read the elements of your collection. The **Iterator** pattern allows to remove some of the elements of your collection while you are iterating over them. If you need to do that, then you want to use the **Iterator** pattern.

**Using an Iterator on a Collection:**

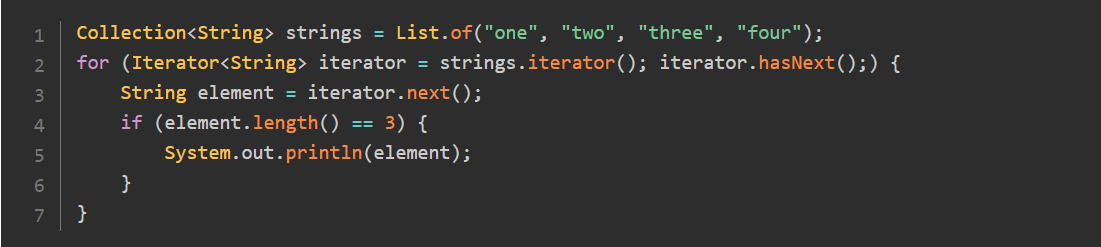
Iterating over the elements of a **collection** uses a special object, an instance of the **Iterator** interface. You can get an **Iterator** object from any extension of the **Collection** interface. The **iterator()** method is defined on the **Iterable interface**, extended by the **Collection interface**, and further extended by all the interfaces of the **collection hierarchy**.

Iterating over the elements of a collection using this object is a two-steps process.

1. First you need to check if there are more elements to be visited with the **hasNext()** method
2. Then you can advance to the next element with the **next()** method.

If you call the **next()** method but there are no more elements in the collection, you will get a **NoSuchElementException**. Calling **hasNext()** is not mandatory, it is there to help you to make sure that there is indeed a next element.

Here is the pattern:



This code produces the following result:

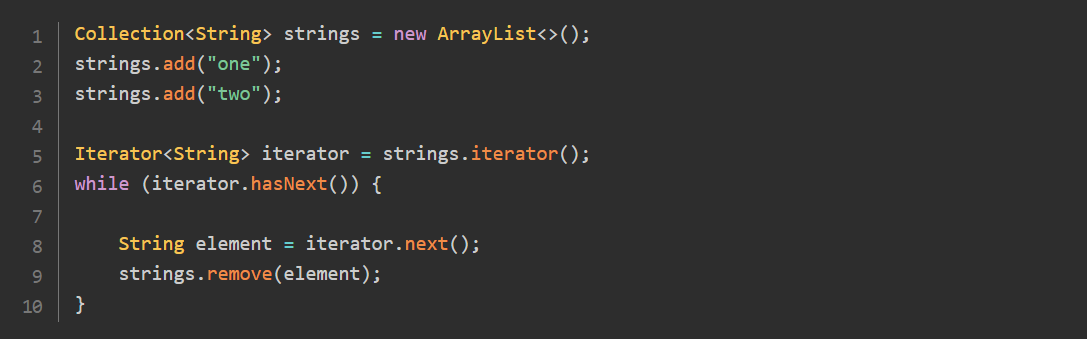


The **Iterator** interface has a third method: **remove().** Calling this method removes the current element from the **collection**. There are cases though where this method is not supported, it will throw an **UnsupportedOperationException**. Quite obviously, calling **remove()** on an immutable collection cannot work, so this is one of the cases. The implementation of Iterator you get from **ArrayList**, **LinkedList** and **HashSet** all support this **remove operation**.

**Updating a Collection While Iterating over It:**

If you happen to modify the content of a collection while iterating over it, you may get a **ConcurrentModificationException**. Getting this exception may be a little confusing, because this exception is also used in concurrent programming. In the context of the Collections Framework, you may get it without touching multithreaded programming.

The following code throws a **ConcurrentModificationException**.



If what you need is to remove the elements of a collection that satisfy a given criteria, you may use the **removeIf()** method.

**Implementing the Iterable Interface:**

Now that you saw what an iterator is in the Collection Framework, you can create a simple implementation of the **Iterable** interface.

Suppose you need to create a **Range** class that models a range of integers between two limits. All you need to do is iterate from the first integer to the last one.

You can implement the **Iterable** interface with a record, a feature introduced in Java SE 16:



You can do the same with a plain class, in case your application does not support Java SE 16 yet. Note that the code of the implementation of **Iterator** is exactly the same.



In both cases, you can use an instance of Range in a for-each statement, since it implements **Iterable**:



Running this code gives you the following result:



**Extending Collection with List:**

**Exploring the List Interface:**

The List interface brings two new functionalities to plain collections.

* This order in which you iterate over the elements of a list is always the same, and it respects the order in which the elements have been added to this list.
* The elements of a list have an index.

**Choosing your Implementation of the List Interface:**

While the **Collection interface** as no specific implementation in the Collections Framework (it relies on the implementations of its sub-interfaces), the **List** interface has 2: **ArrayList** and **LinkedList**. As you may guess, the first one is built on an **internal array**, and the second on a **doubly-linked list**.

Is one of these implementation better than the other? If you are not sure which one to choose, then your best choice is probably **ArrayList**.

What was true for **linked lists** when computing was invented in the 60's does not hold anymore, and the capacity of **linked lists** to outperform arrays on **insertion** and **deletion** operations is greatly diminished by modern **hardware**, **CPU caches**, and **pointer chasing**. Iterating over the elements of an **ArrayList** is much faster that over the elements of a **LinkedList**, mainly due to pointer chasing and CPU cache misses.

There are still cases where a **linked** list is faster than an **array**. A **doubly-linked list** can access its first and last element faster than an **ArrayList** can. This is the main use case that makes **LinkedList** better than **ArrayList**. So if your application needs a **Last In, First Out** (LIFO, covered later in this tutorial) **stack**, or a **First In, First Out** (FIFO, also covered later) waiting **queue**, then choosing a **linked list** is probably your best choice.

On the other hand, if you plan to iterate through the elements of your **list**, or to access them **randomly** by their index, then the **ArrayList** is probably your best bet.

**Accessing the Elements Using an Index:**

The **List** interface brings several methods to the **Collection** interface, that deal with indexes.

**Accessing a Single Object:**

* **add(index, element):** inserts the given object at the index, adjusting the index if there are remaining elements
* **get(index):** returns the object at the given index
* **set(index, element):** replaces the element at the given index with the new element
* **remove(index):** removes the element at the given index, adjusting the index of the remaining elements.

Calling these methods work only for valid indexes. If the given index is not valid then an **IndexOutOfBoundsException** exception will be thrown.

**Finding the Index of an Object:**

The methods **indexOf(element)** and **lastIndexOf(element)** return the index of the given element in the list, or -1 if the element is not found.

**Getting a SubList:**

The **subList(start, end)** returns a list consisting of the elements between indexes **start and end - 1**. If the indexes are invalid then an I**ndexOutOfBoundsException** exception will be thrown.

Note that the returned list is a view on the main list. Thus, any modification operation on the **sublist** is reflected on the main list and vice-versa.

**For instance, you can clear a portion of the content of a list with the following pattern:**



Running this code gives you the following result:



**Inserting a Collection:**

The last pattern of this list is about inserting a collection at a given indexes: **addAll(int index, Collection collection).**

**Sorting the Elements of a List:**

A list keeps its elements in a known order. This is the main difference with a plain collection. So it makes sense to sort the elements of a list. This is the reason why a **sort()** method has been added to the **List** interface in JDK 8.

In Java SE 7 and earlier, you could sort the elements of your **List** by calling **Collections.sort()** and pass you list as an argument, along with a **comparator** if needed.

Starting with Java SE 8 you can call **sort()** directly on your list and pass your comparator as an argument. There is no overload of this method that does not take any argument. Calling it with a **null comparator** will assume that the elements of your **List** implement **Comparable**, you will get a **ClassCastException** if this is not the case.

If you do not like calling methods will **null arguments** (and you are right!), you can still call it with **Comparator.naturalOrder()** to achieve the same result.

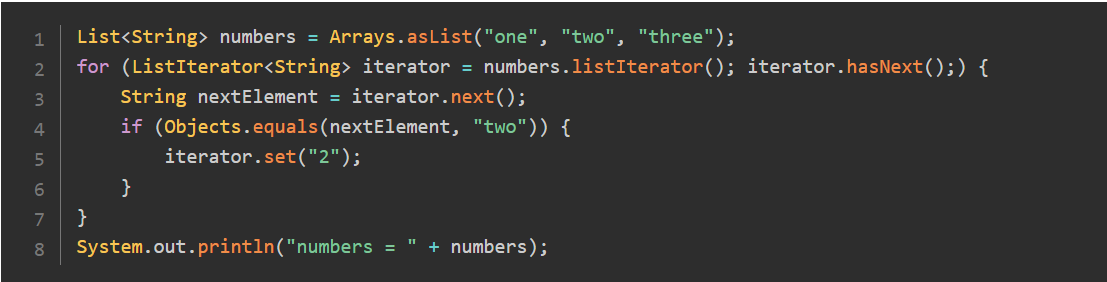
**Iterating over the Elements of a List:**

The List interface gives you one more way to **iterate** over its elements with the **ListIterator**. You can get such an iterator by calling **listIterator().** You can call this method with no argument, or pass an **integer** **index** to it. In that case, the iteration will **start** at this index.

The **ListIterator** interface extends the regular **Iterator** that you already know. It adds several methods to it.

* **hasPrevious()** and **previous():** to iterate in the descending order rather than the ascending order
* **nextIndex()** and **previousIndex():** to get the index of the element that will be returned by the next **next()** call, or the **next previous()** call
* **set(element):** to update the last element returned by **next()** or **previous().** If neither of these methods have been called on this iterator then an **IllegalStateException** is raised.

Let us see this **set()** method in action:



Running this code will give you the following result:



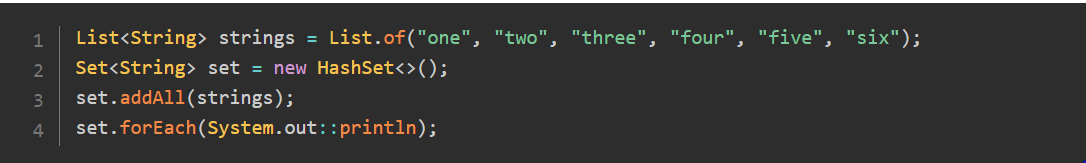
**Extending Collection with Set, SortedSet and NavigableSet:**

**Exploring the Set Interface:**

The **Set** interface does not bring any new method to the **Collection** interface. The Collections Framework gives you one plain implementation of the **Set** interface: **HashSet**. Internally, a **HashSet** wraps an instance of **HashMap**, a class that will be covered later, that acts as a delegate for **HashSet**.

As you already saw, what a **Set** brings to a Collection is that it forbids duplicates. What you lose over the **List** interface is that your elements are stored in no particular order. There is very little chance that you will iterate over them in the same order as you added them to your **set**.

You can see this in the following example:



Running this code will produce the following:



Some implementations of **Set** give you the same order when you iterate over their elements, but since this is not guaranteed, your code should not rely on that.

**Extending Set with SortedSet:**

The first extension of **Set** is the **SortedSet** interface. The **SortedSet** interface keeps its elements sorted according to a certain **comparison logic**. The Collections Framework gives you one implementation of **SortedSet**, called **TreeSet**.

As you already saw, either you provide a **comparator** when you build a **TreeSet**, or you implement the Comparable interface for the elements you put in the **TreeSet**. If you do both, then the **comparator** takes **precedence**.

The **SortedSet** interface adds new methods to **Set**.

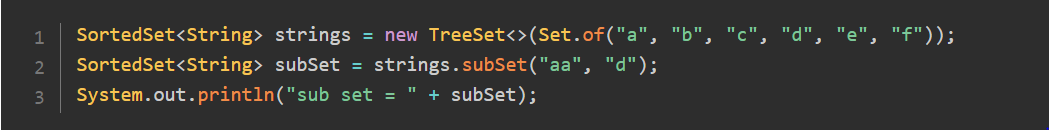
**first()** and **last()** returns the **lowest** and the **largest** elements of the **set**

**headSet(toElement)** and **tailSet(fromElement)** returns you subsets containing the elements lower than **toElement** or greater than **fromElement**

**subSet(fromElement, toElement)** gives you a subset of the element between **fromElement** and **toElement**.

The **toElement** and **fromElement** do not have to be elements of the main set. If they are, then **toElement** is not included in the result and **fromElement** is, following the usual convention.

Consider the following example:



Running this code will give you the following:



The three subsets that these methods return are views on the main set. No copy is made, meaning that any change you make to these subsets will be reflected in the set, and the other way round.

You can remove or add elements to the main set through these subsets. There is one point you need to keep in mind though. These three subsets remember the limits on which they have been built. For consistency reasons, it is not legal to add an element through a subset outside its limits. For instance, if you take a **headSet** and try to add an element greater than **toElement**, then you will get an **IllegalArgumentException**.

**Extending SortedSet with NavigableSet:**

Java SE 6 saw the introduction of an extension of **SortedSet** with the addition of more methods. It turns out that the **TreeSet** class was retrofitted to implement **NavigableSet**. So you can use the same class for both interfaces.

Some methods are overloaded by **NavigableSet**.

* **headSet()**, **tailSet(),** and **subSet()** may take a further **boolean** arguments to specify whether the limits (**toElement** or **fromElement**) are to be included in the resulting subset.

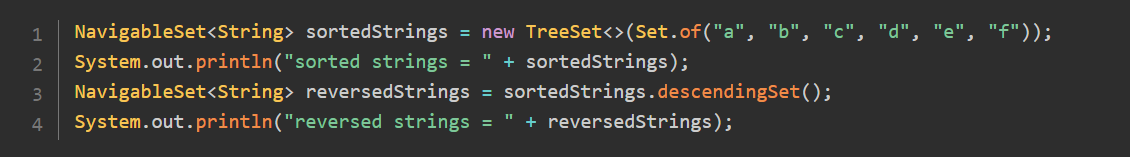
Other methods have been added.

* **ceiling(element),** and **floor(element)** return the greatest element lesser or equal than, or the lowest element greater or equal than the provided element. If there is no such element then **null** is returned
* **lower(element)**, and **higher(element)** return the greater element lesser than, or the lowest element greater than the provided element. If there is no such element then **null** is returned.
* **pollFirst(),** and **pollLast()** return and removes the lowest or the greatest element of the set.
* Furthermore, **NavigableSet** also allows you to **iterate** over its elements in **descending** order. There are two ways to do this.

You can call **descendingIterator():** it gives you a regular Iterator that traverses the set in the **descending** order.

You can also call **descendingSet().** What you get in return is another **NavigableSet** that is a view on this set and that makes you think you have the **same set**, sorted in the **reversed order**.

The following example demonstrates this.



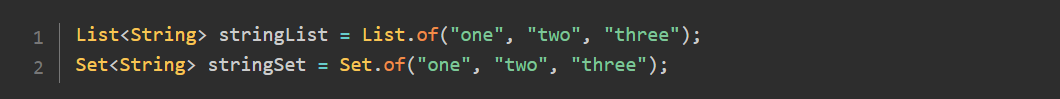
Running this code will give you the following:



**Creating and Processing Data with the Collections Factory Methods:**

**Creating Immutable Collections:**

Java SE 9 saw the addition of a set of factory methods to the **List** and **Set** interfaces to create lists and sets. The pattern is very simple: just call the **List.of()** or **Set.of()** static method, pass the elements of your list and set, and that's it.



Several points are worth noting though.

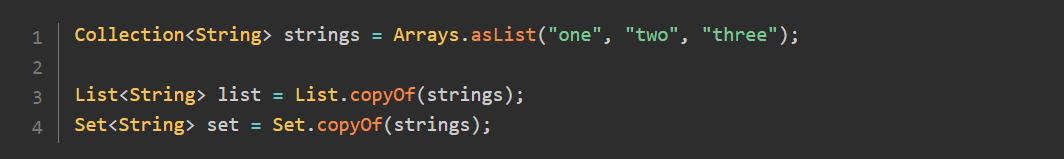
* The implementation you get in return may vary with the number of elements you put in your list or set. None of them is **ArrayList** or HashSet, so your code should not rely on that.
* Both the list and the **set** you get are **immutable structures**. You cannot **add** or **modify** elements in them, and you cannot modify these elements. If the **objects** of these structures are **mutable**, you can still mutate them.
* These structures do not accept **null** values. If you try to add a **null** value in such a **list** or **set**, you will get an exception.
* The **Set** interface does not allow **duplicates**: this is what a **set** is about. Because it would not make sense to create such a set with duplicate values, it is assumed that writing such a code is a bug. So you will get an exception if you try to do that.
* The implementations you get are **Serializable**.

These **of()** methods are commonly referred to as convenience factory methods for collections.

**Getting an Immutable Copy of a Collection:**

Following the success of the convenience factory methods for collections, another set of convenience methods have been added in Java SE 10 to create immutable copies of collections.

There are two of them: **List.copyOf()** and **Set.copyOf().** They both follow the same pattern:



In all cases, the collection you need to copy should not be null and should not contain any null elements. If this collection has duplicates, only one of these elements will be kept in the case of **Set.copyOf().**

What you get in return is an immutable copy of the collection passed as an argument. So modifying this collection will not be reflected in the list or set you get as a copy.

None of the implementations you get accept **null** values. If you try to copy a collection with **null** values, you will get a **NullPointerException.**

**Wrapping an Array in a List:**

The **Collections** Framework has a class called **Arrays** with about **200** methods to handle arrays. Most of them are implementing various algorithms on arrays, like **sorting**, **merging**, **searching**, and are not covered in this section.

There is one though that is worth mentioning: **Arrays.asList().** This method takes a vararg as an argument and returns a **List** of the elements you **passed**, preserving their order. This method is not part of the convenience factory methods for collections but is still very useful.

This **List** acts as a wrapper on an **array**, and behaves in the same way, which maybe a little confusing at first. Once you have **set** the **size** of an array, you cannot change it. It means that you cannot **add** an element to an existing array, nor can you **remove** an element from it. All you can do is **replace** an existing element with another one, possibly **null**.

The List you get by calling **Arrays.asList()** does exactly this.

If you try to **add** or **remove** an element, you will get an **UnsupportedOperationException**, whether you do that directly or through the **iterator**.

Replacing existing elements is **OK**.

So this list is not immutable, but there are restrictions on how you can change it.

**Using the Collections Factory Class to Process a Collection:**

The Collections Framework comes with another factory class: **Collections**, with a set of method to manipulate collections and their content. There are about 70 methods in this class, it would be tedious so examine them one-by-one, so let us present a subset of them.

**Extracting the Minimum or the Maximum from a Collection:**

The Collections class give you two methods for that: the **min()** and the **max().** Both methods take the collection as an argument from which the min or the max is extracted. Both methods have an overload that also takes a comparator as a further argument.

If no comparator is provided then the elements of the collection must implement **Comparable**. If not, a **ClassCastException** will be raised. If a comparator is provided, then it will be used to get the **min** or the **max**, whether the elements of the collection are comparable or not.

Getting the **min** or the **max** of an empty collection with this method will raise a **NoSuchMethodException**.

**Finding a Sublist in a List:**

Two methods locate a given sublist in a bigger list:

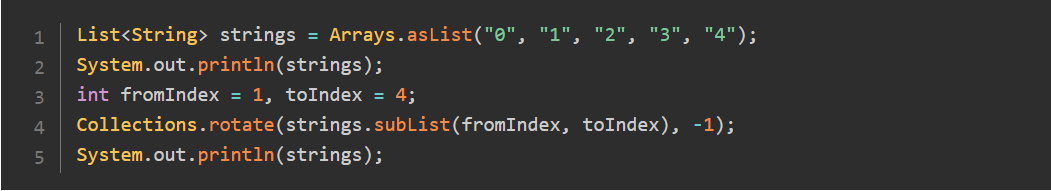
**indexOfSublist(List<?> source, List<?> target):** returns the first index of the first element of the target list in the source list, or -1 if it does not exist;

**lastIndexOfSublist(List<?> source, List<?> target):** return the last of these indexes.

**Changing the Order of the Elements of a List:**

**Several methods can change the order of the elements of a list:**

* **sort()** sorts the list in place. This method may take a comparator as an argument. As usual, if no comparator is provided, then the elements of the list must be comparable. If a comparator is provided, then it will be used to compare the elements. Starting with Java SE 8, you should favor the **sort()** method from the **Listinterface**.
* **shuffle()** randomly shuffles the elements of the provided **list**. You can provide your instance of Random if you need a random shuffling that you can repeat.
* **rotate()** rotates the elements of the **list**. After a rotation the element at **index 0** will be found at **index 1** and so on. The last elements will be moved to the first place of the list. You can combine **subList()** and **rotate()** to remove an element at a given index and to insert it in another place in the **list**. This can be done with the following code:



The result is the following:



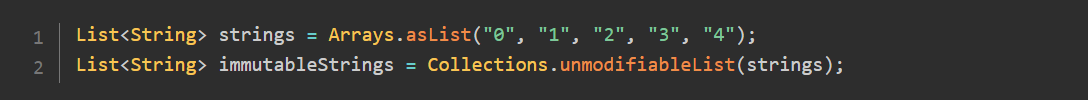
The element at index **fromIndex** has been removed from its place, the list has been reorganized accordingly, and the element has been inserted at index **toIndex - 1**.

* **reverse():** reverse the order of the elements of the list.
* **swap():** swaps two elements from the list. This method can take a list as an argument, as well as a plain array.

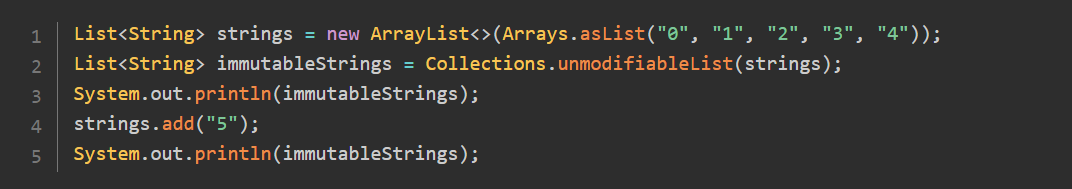
**Wrapping a Collection in an Immutable Collection:**

The **Collections** factory class gives you several methods to create immutable wrappers for your collections or maps. The content of the structure is not duplicated; what you get is a wrapper around your structure. All the attempts to modify it will raise exceptions.

All these methods starts with **unmodifiable**, followed by the name of the type of your structure. For instance, to create an immutable wrapper of a list, you can call:



Just a word of warning: it is not possible to modify your collection through this wrapper. But this wrapper is backed by your collection, so if you modify it by another means, this modification will be reflected in the immutable collection. Let us see that in the following code:



Running this example will give you the following:



If you plan to create an immutable collection using this pattern, defensively copying it first may be a safe precaution.

**Wrapping a Collection in a Synchronized Collection:**

In the same way as you can create immutable wrappers for your maps and collections, the **Collections** factory class can create synchronized wrappers for them. The patterns follow the same naming convention as the names for methods that create immutable wrappers: the methods are called **synchronized** followed by **Collection**, **List**, **Set**, etc...

There are two precautions you need to follow.

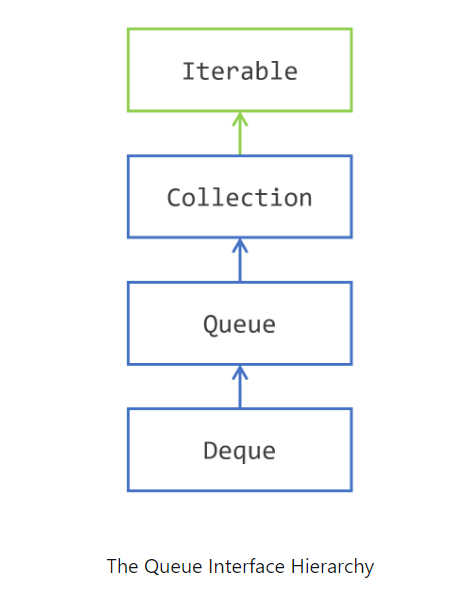
* All the accesses to your collection should be made through the wrapper you get
* Traversing your collection with an iterator or a stream should be synchronized by the calling code on the list itself.
* Not following these rules will expose your code to race conditions.

Synchronizing collections using the **Collections** factory methods may not be your best choice. The Java Util Concurrent framework has better solutions to offer.

**Storing Elements in Stacks and Queues:**

**Finding Your Way in the Queue Hierarchy:**

Java SE 5 saw the addition of a new interface in the Collections Framework: the **Queue** interface, further extended in Java SE 6 by the **Deque** interface. The **Queue** interface is an extension of the **Collection** interface.



**Pushing, Popping and Peeking:**

The stack and queue structures are classic data structures in computing. Stacks are also called LIFO stacks, where **LIFO** stands for Last In, First Out. Queues are known as **FIFO**: First In First Out.

These structures are very simple and gives you three main operations.

* **push(element):** adds an element to the queue, or the stack
* **pop():** removes an element from the stack, that is, the youngest element added
* **poll():** removes an element from the queue, that is, the oldest element added
* **peek():** allows you to see the element you will get with a pop() or a poll(), but without removing it from the queue of the stack.

There are two reasons to explain the success of these structures in computing. The first one is their simplicity. Even in the very early days of computing, implementing these was simple. The second one is their usefulness. Many algorithms use stacks for their implementations.

**Modeling Queues and Stacks:**

The Collections Framework gives you two interfaces to model queues and stacks:

* the **Queue** interface models a queue;
* the **Deque** interface models a double ended queue (thus the name). You can push, pop, poll and peek elements on both the tail and the head of a Deque, making it both a queue and a stack.

Stacks and queues are also widely used in concurrent programming. These interfaces are further extended by more interfaces, adding methods useful in this field. These interfaces, **BlockingQueue**, **BlockingDeque** and **TransferQueue**, are at the intersection of the Collections Framework and concurrent programming in Java, which is outside the scope of this tutorial.

Both the **Queue** and the **Deque** interfaces add behavior to these three fundamental operations to deal with two corner cases.

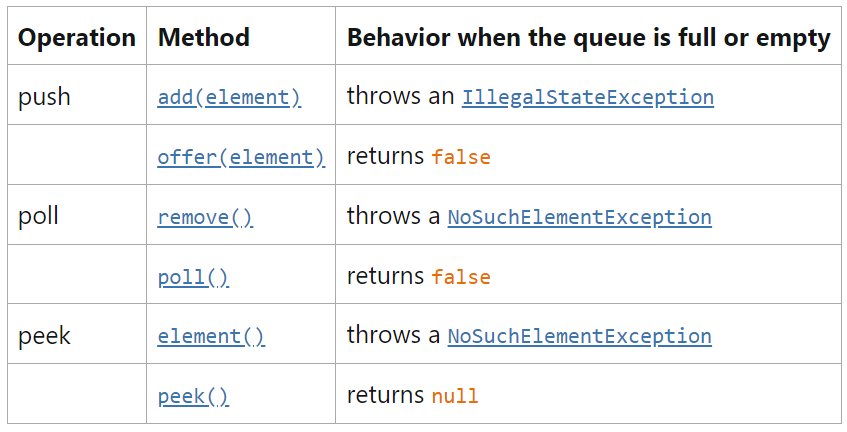
* A queue may be full and not able to accept more elements
* A queue may be empty and cannot return an element with a pop, poll, nor the peek operation.

In fact this question needs to be answered: how does an implementation should behave in these two cases?

**Modeling FIFO Queues with Queue:**

The **Queue** interface gives you two ways of dealing with these corner cases. An exception can be thrown, or a special value can be returned.

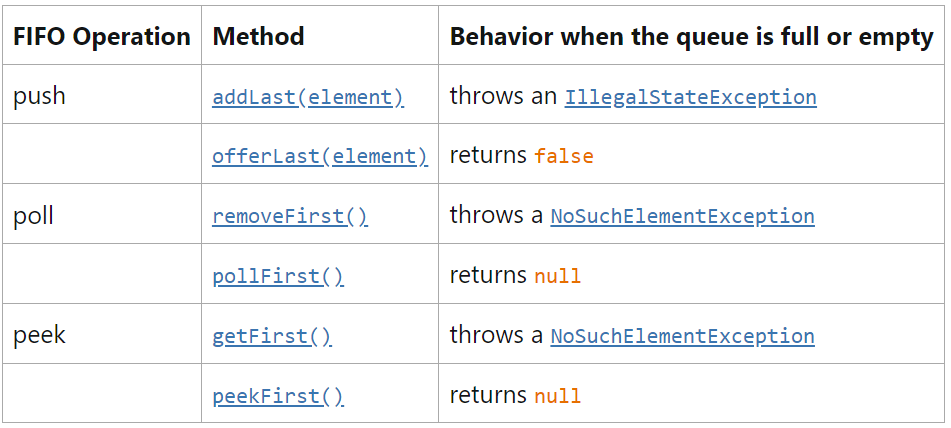
Here is the table of the methods **Queue** gives you.



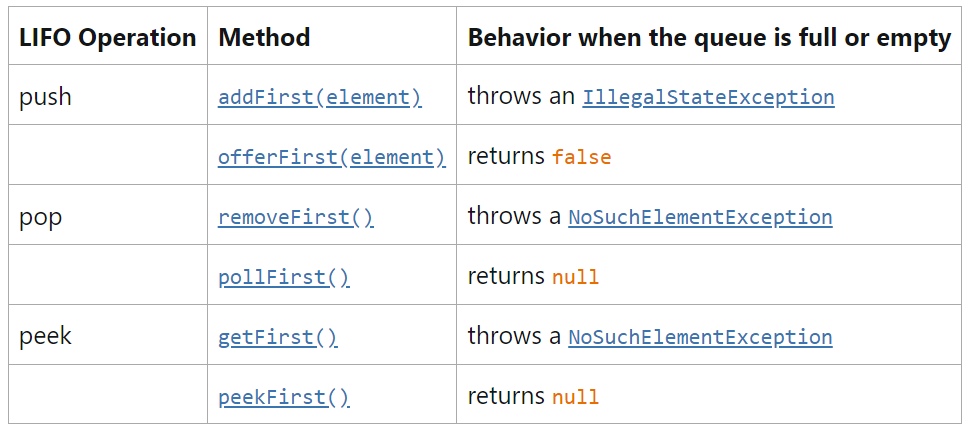
**Modeling LIFO Stacks and FIFO Queues with Deque:**

Java SE 6 added the **Deque** interface as an extension of the **Queue** interface. Of course, the methods defined in **Queue** are still available in **Deque**, but **Deque** brought a new naming convention. So these methods have been **duplicated** in **Deque**, following this new naming convention.

Here is the table of the methods defined in **Deque** for the **FIFO** operations.



And here is the table of the methods defined in **Deque** for the LIFO operations.



The **Deque** naming convention is straightforward and is the same as the one followed in the **Queue** interface. There is one difference though: the **peek** operations are named **getFirst()** and **getLast()** in **Deque**, and **element()** in Queue.

Moreover, **Deque** also defines the methods you would expect in any queue or stack class:

* **push(element):** adds the given element to the head of the double ended queue
* **pop():** removes and return the element at the head of the double ended queue
* **poll():** does the same at the tail of the double ended queue
* **peek():** shows you the element at the tail of the double ended queue.

In case there is no element to pop, poll, or peek, then a null value is returned by these methods.

**Implementing Queue and Deque:**

The Collections Framework gives you three implementations of **Queue** and **Deque**, outside the concurrent programming space:

* **ArrayDeque**: which implements both. This implementation is backed by an array. The capacity of this class automatically grows as elements are added. So this implementation always accepts new elements.
* **LinkedList**: which also implements both. This implementation is backed by a linked list, making the access to its first and last element very efficient. A **LinkedList** will always accept new elements.
* **PriorityQueue**: that only implements Queue. This **queue** is backed by an array that keeps its elements sorted by their natural order or by an order specified by a **Comparator**. The head of this queue is always the least element of the queue with respect to the specified ordering. The capacity of this class automatically grows as elements are added.

**Staying Away from the Stack Class:**

It may seem tempting to use the **Stack** class offered by the JDK. This class is simple to use and to understand. It has the three expected methods **push(element),** **pop(),** and **peek(),** and seeing this class referenced in your code makes it perfectly readable.

It turns out that this class is an extension of the **Vector class**. Back in the days before the Collections Framework was introduced, Vector was your best choice to work with a list. Although Vector is not deprecated, its usage is discouraged. So is the usage of the **Stack** class.

The **Vector** class is thread safe, and so is **Stack**. If you do not need the thread safety, then you can safely replace its usage with Deque and **ArrayDeque**. If what you need is a thread-safe stack, then you should explore the implementations of the **BlockingQueue** interface.

**Using Maps to Store Key Value Pairs:**

**Introducing the Map Hierarchy:**

The second main structure offered by the Collections Framework is an implementation of a very classic data structure: the **hashmap** structure. This concept is not new and is fundamental in structuring data, whether in-memory or not. How does it work and how has it been implemented in the Collections Framework?

A **hashmap** is a structure able to store **key-value** pairs. The value is any object your application needs to handle, and a key is something that can represent this object.

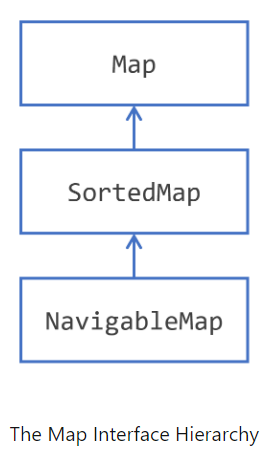
Suppose you need to create an application that has to handle invoices, represented by instances of an **Invoice** class. Then your values are these Invoice instances, and your keys could be the invoice numbers. Each invoice has a number, and that number is unique among all your invoices.

Generally speaking, each value is bound to a key, just as an **invoice** is bound to its invoice number. If you have a given key, you can retrieve the value. Usually a key is a simple object: think of a string of several characters or a number. The value, on the other hand, can be as complex as you need. This is what **hashmaps** have been made for: you can manipulate keys, move them from one part of your application to another, transmit them over a network, and when you need the full object, then you can retrieve it with its key.

Before you see all the details of the **Map** interface, here are the notions you need to have in mind.

* A hashmap can store key-value pairs
* A key acts as a symbol for a given value
* A key is a simple object, a value can be as complex as needed
* A key is unique in a hashmap, a value does not have to be unique
* Every value stored in a hashmap has to be bound to a key, a key-value pair in a map forms an entry of that map
* A key can be used to retrieve its bound value.

The Collections Framework gives you a **Map** interface that implements this concept, along with two extensions, **SortedMap** and **NavigableMap**, as shown on the following figure.



This hierarchy is very simple and looks like the **Set** hierarchy, with **SortedSet** and **NavigableSet**. Indeed, a **SortedMap** shares the same kind of semantics as the **SortedSet**: a **SortedMap** is a map that keeps its **key-value** pairs sorted by their keys. The same goes for **NavigableMap**: the methods added by this interface are the same kind of methods than the ones added by **NavigableSet** to **SortedSet**.

The JDK gives you several implementations of the **Map** interface, the most widely used is the **HashMap** class.

Here are the two other implementations.

* **LinkedHashMap** is a **HashMap** with an internal structure to keep the **key-value** pairs ordered. **Iterating** on the keys or the key-value pairs will follow the order in which you have added your key-value pairs.
* **IdentityHashMap** is a specialized Map that you should only be used in very precise cases. This implementation is not meant to be generally used in application. Instead of using **equals()** and **hashCode()** to compare the key objects, this implementation only compares the references to these keys, with an equality operator **(==).** Use it with caution, only if you are sure this is what you need.

You may have heard of **multimaps**. Multimap is a concept where a **single key** can be associated to more than **one** value. This concept is not directly supported in the **Collections** Framework. This feature may be useful though, and you will see later in this tutorial how you can create **maps** with values that are in fact **lists** of values. This pattern allows you to create **multimap-like structures**.

**Using the Convenience Factory Methods for Collections to Create Maps:**

As you already saw, Java SE 9 added methods to the **List** and **Set** interfaces to create immutable lists and sets.

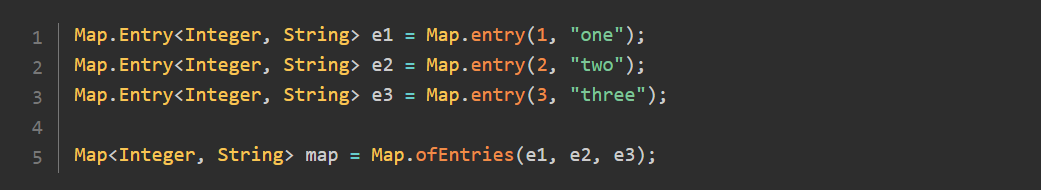
There are such methods on the **Map** interface that create immutable maps and immutable entries.

You can create a Map easily with the following pattern.



There is one caveat though: you can only use this pattern it if you have no more than 10 key-value pairs.

If you have more, then you need to use another pattern:



You can also write this pattern in this way, and use static imports to further improve its readability.



There are restrictions on these maps and entries created by these factory methods, as for the sets:

* The map and the entries you get are immutable objects
* Null entries, **null keys**, and **null** values are not allowed
* Trying to create a map with duplicate keys in this way does not make sense, so as a warning you will get an **IllegalArgumentException** at map creation.

**Storing Key/Value Pairs in a Map:**

The relationship between a key and its bound value follows these two simple rules.

* A key can be bound to only one value
* A value can be bound to several keys.

This leads to several consequences for the content of the map.

* The set of all the keys cannot have any duplicates, so it has the structure of a **Set**
* The set of all the key/value pairs cannot have duplicates either, so it also has the structure of a **Set**
* The set of all the values may have duplicates, so it has the structure of a plain **Collection**.

Then, you can define the following operations on a map:

* Putting a key/value pair in the map. This may fail if the key is already defined in the map
* Getting a value from a key
* Removing a key from a map, along with its value.

You can also define the classic, set-like operations:

* Checking if the map is empty or not
* Getting the number of key-value pairs contained in the map
* Putting all the content of another map in this map
* Clearing the content of a map.

All these operations and concepts are implemented in the **Map** interface, along with some others that you are going to see in the following.

**Exploring the Map interface:**

The **Map** interface is the base type that models the notion of map in the JDK.

You should be extremely careful when choosing the type of the keys for your maps. In a nutshell, choosing a mutable key is not prohibited but is dangerous and discouraged. Once a key has been added to a map, mutating it may lead to changing its hash code value, and its identity. This may make your key-value pair unrecoverable or may get you a different value when querying your map. You will see this later on an example.

The **Map** defines a member interface: **Map.Entry** to model a key-value pair. This interface defines three methods to access the key and the values:

* **getKey():** to read the key;
* **getValue()** and **setValue**(value): to read and update the value bound to that key.

The **Map.Entry** objects you can get from a given map are views on the content of the map. Modifying the value of an entry object is thus reflected in the map and the other way round. This is the reason why you cannot change the key in this object: it could corrupt your map.

The Map interface is the base type that models the notion of map in the JDK.

**Managing the Content of a Map:**

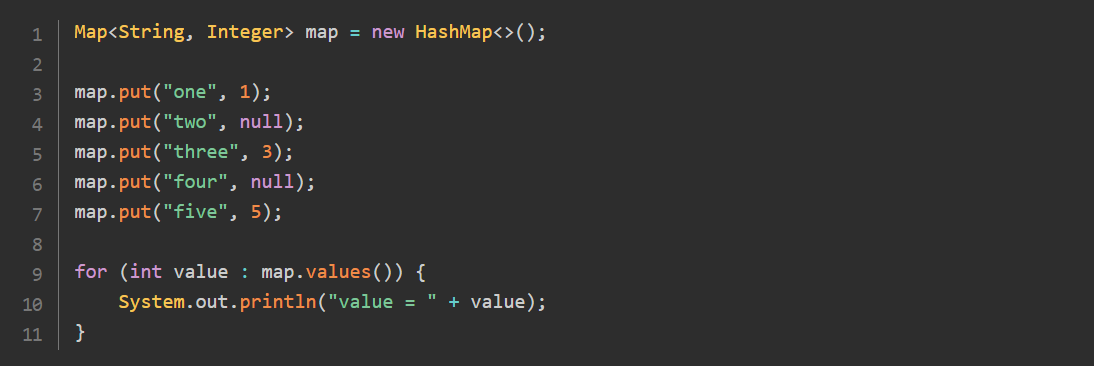
**Adding a Key Value Pair to a Map:**

You can simply add a key/value pair in a map with **put(key, value).** If the key is not already present in the map, then the key/value pair is simply added to the map. If it is, then the existing value is replaced with the new one.

In both cases, the **put()** method returns the existing value currently bound to the key. This means that if this a new key, a call to **put()** will return null.

Java SE 8 introduces the **putIfAbsent()** method. This method can also add a key/value pair to the map, only if the key is not already present and not associated to a null value. This may seem a bit confusing at first, but **putIfAbsent()** will replace a null value with the new value provided.

This method is very handy if you need to get rid of faulty null values in your map. For instance the following code will fail with a **NullPointerException** because you cannot auto-unbox a null **Integer** to an int value.



If you take a close look at this code, you will see that **map.values()** is a Collection<Integer>. So iterating on this collection produces instances of **Integer**. Because you declared value as an int, the compiler will auto-unbox this Integer to an int value. This mechanism fails with a **NullPointerException** if the instance of **Integer** is **null**.

You may fix this map with the following code, which replaces the faulty null values with a default value, **-1**, that will not generate any **NullPointerException** anymore.



Running the previous code will print the following. As you can see this map does not contain any null values anymore:

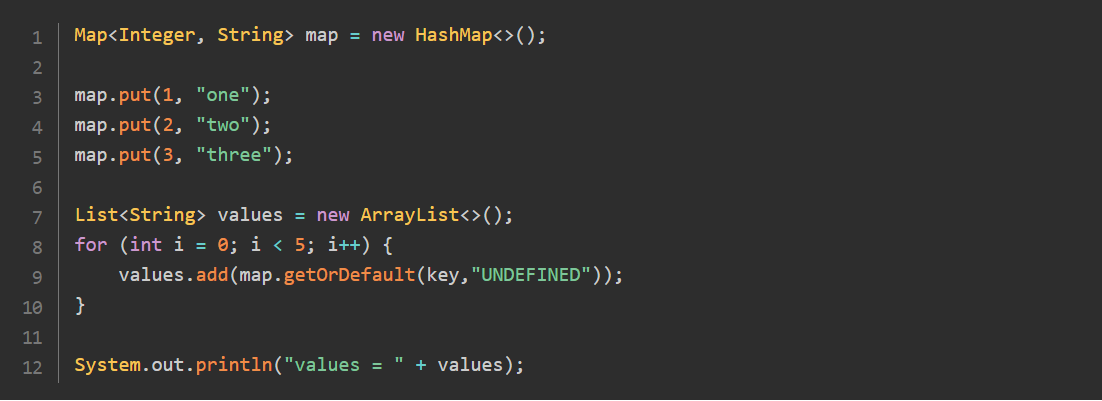


**Getting a Value From a Key:**

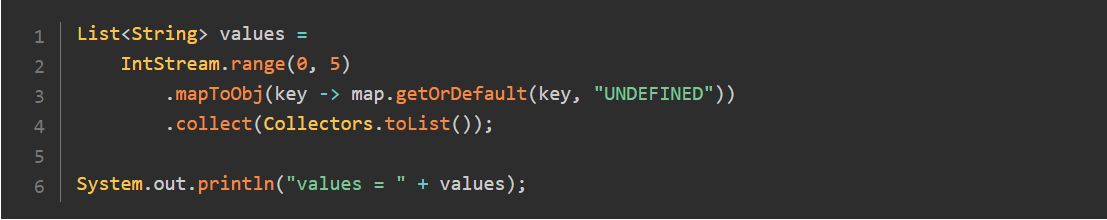
You can get a value bound to a given key simply by calling the **get(key)** method.

Java SE 8 introduced the **getOrDefault()** method that takes a key and a default value which is returned if the key is not in the map.

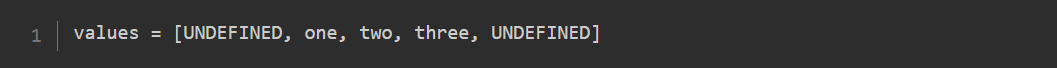
Let us see this method in action in an example:



Or, if you are familiar with streams (which are covered later in this tutorial):



Both codes print out the same result:



**Removing a Key from a Map:**

Removing a key/value pair is made by calling the **remove(key)** method. This method returns the value that was bound to that key, so it may return **null**.

It may be risky to blindly **remove** a key/value pair from a **map** if you do not know the value that is bound to that key. Thus, Java SE 8 added an overload that takes a value as a second argument. This time, the key/value pair is removed only if it fully matches the key/value pair in the map.

This **remove(key, value)** method returns a **boolean** value, true if the key/value pair was removed from the map.

**Checking for the Presence of a Key or a Value:**

You have two methods to check for the presence of a given key or a given value: **containsKey(key)** and **containsValue(value).** Both methods return true if the map contains the given key or value.

**Checking for the Content of a Map:**

The **Map** interface also brings methods that look like the ones you have in the **Collection** interface. These methods are self-explanatory: **isEmpty()** returns true for empty maps, **size()** returns the number of key/value pairs, and **clear()** removes all the content of the map.

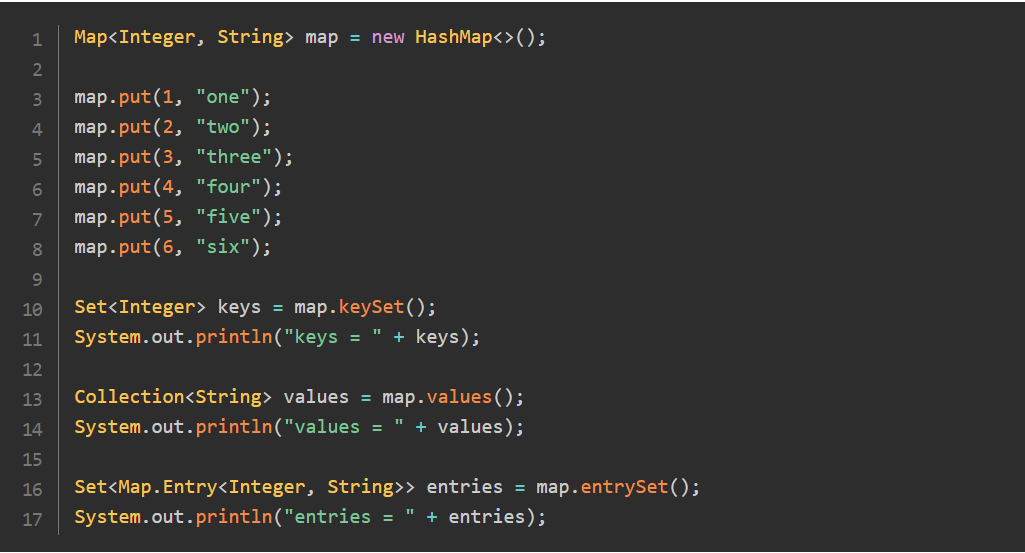
There is also a method to add the content of a given map to the current map: **putAll(otherMap).** If some keys are present in both maps, then the values of **otherMap** will erase those of this map.

**Getting a View on the Keys, the Values or the Entries of a Map:**

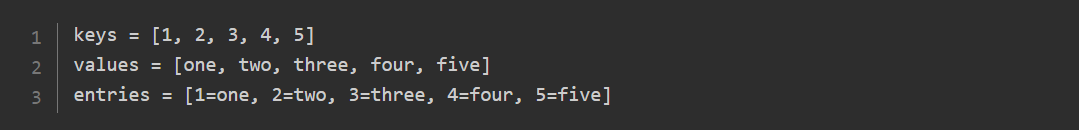
You can also get different sets defined on a map.

* **keySet():** returns an instance of **Set**, containing the keys defined in the map
* **entrySet():** returns an instance of **Set<Map.Entry>,** containing the key/value pairs contained in the map
* **values():** returns an instance of **Collection**, containing the values present in the map.

The following examples show these three methods in action:



Running this code produces the following result:

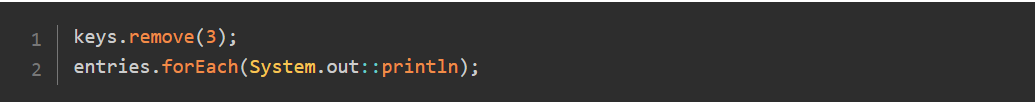


These sets are views backed by the current map. Any change made to the map is reflected in those views.

**Removing a Key From the Set of Keys:**

Modifying one of these sets will also be reflected in the map: for instance, removing a key from the set returned by a call to **keySet()** removes the corresponding key/value pair from the map.

For instance, you can run this code on the previous map:



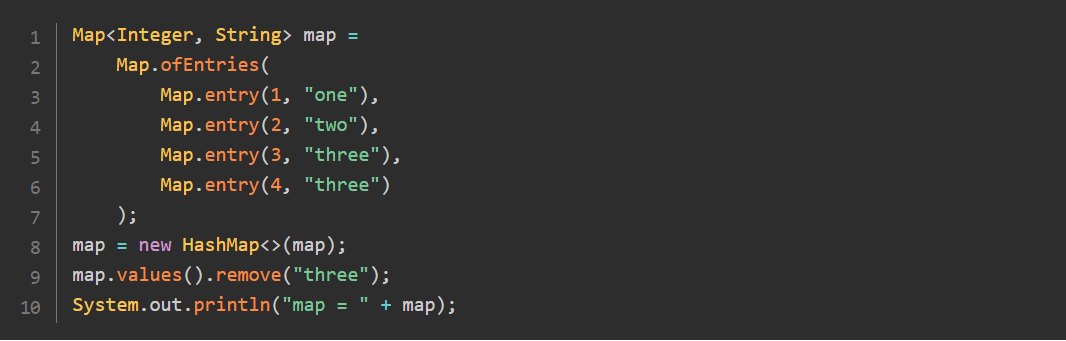
It will produce the following resut:



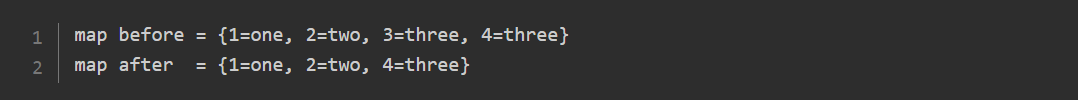
**Removing a Value From the Collection of Values:**

Removing a value is not as simple because a value can be found more than once in a map. In that case, removing a value from the collection of values just removes the first matching key/value pair.

You can see that on the following example.



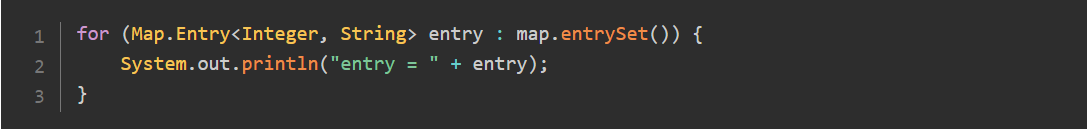
Running this code will produce the following result.



As you can see, only the first key/value pair has been removed in this example. You need to be careful in this case because if the implementation you chose is a **HashMap**, you cannot tell in advance what key/value pair will be found.

You do not have access to all the operations on these sets though. For instance, you cannot add an element to the set of keys, or to the collection of values. If you try to do that, you will get an **UnsupportedOperationException**.

If what you need is to iterate over the key/value pairs of a map, then your best choice is to iterate directly on the set of key/value pairs. It is much more efficient to do that, rather than iterating on the set of keys, and getting the corresponding value. The best pattern you can use is the following:



**Handling Map Values with Lambda Expressions:**

**Consuming the Content of a Map:**

The Map interface has a **forEach()** method that works in the same way as the **forEach()** method on the **Iterable** interface. The difference is that this **forEach()** method takes a **BiConsumer** as an argument instead of a **simple Consumer.**

Let us create a simple map and print out its content.

**Replacing Values:**

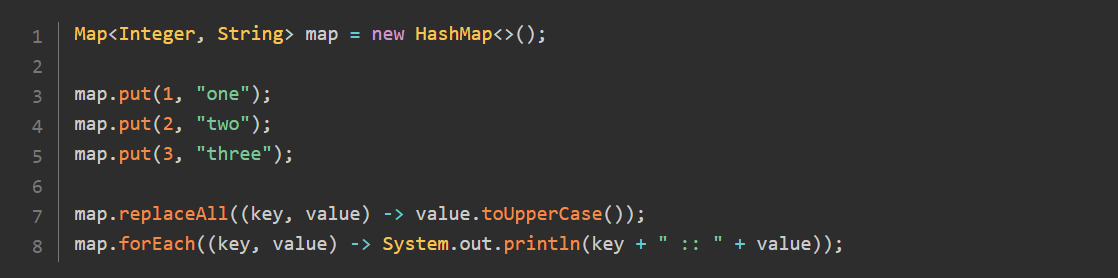
The **Map** interface gives you three methods to replace a value bound to a key with another value.

The first one is **replace(key, value),** which replaces the existing value with the new one, blindly. This is the equivalent of a **put-if-present** operation. This method returns the value that was removed from your map.

If you need finer control, then you can use an overload of this method, which takes the existing value as an **argument**: **replace(key, existingValue, newValue).** In this case, the existing value is replaced only if it matches the **new** value. This method returns true if the **replacement** occurred.

The Map interface has also a method to replace all the values of your map using a **BiFunction**. This **BiFunction** is a remapping function, which takes the key and the value as arguments, and returns a new value, which will replace the existing value. A call to this method iterates internally on all the **key/value** pairs of your map.

The following example shows how you can use this **replaceAll()** method:



Running this code produces the following result:



**Computing Values:**

The **Map** interface gives you a third pattern to add key-value pairs to a map or modify a map's existing values in the form of three methods: **compute(),** **computeIfPresent(),** and **computeIfAbsent().**

These three methods take the following arguments:

* the **key** on which the computation is made
* the **value** bound to that key, in the case of **compute()** and **computeIfPresent()**
* a **BiFunction** that acts as a remapping function, or a mapping function in the case of **computeIfAbsent().**

In the case of **compute(),** the remapping **bi-function** is called with two arguments. The first one is the **key**, and the second one is the existing **value** if there is one, or **null** if there is none. Your remapping bifunction can be called with a null value.

For **computeIfPresent(),** the remapping function is called if there is a value bound to that **key** and if it is not **null**. If the key is bound to a **null** value, then the remapping function is not called. Your remapping function cannot be called with a null value.

For **computeIfAbsent(),** beacause there is no value bound to that key, the remapping function is in fact a simple **Function** that takes the key as an argument. This function is called if the key is not present in the map or if it is bound to a null value.

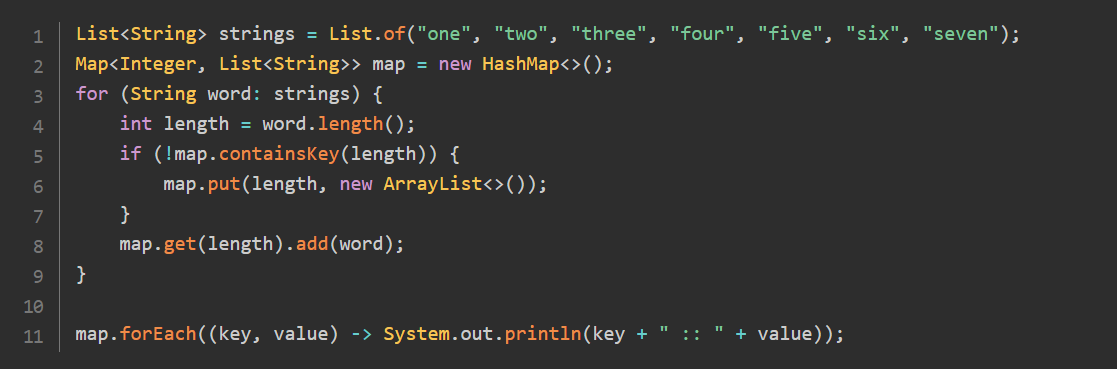
In all cases, if your **bifunction** (or function) returns a **null** value, then the key is removed from the map: no mapping is created for that key. No key/value pair with a null value can be put in the map using one of these three methods.

In all cases, the value returned is the new value bound to that **key** in the map or null if the remapping function returned null. It is worth pointing out that this semantic is different from the **put()** methods. The **put()** methods return the previous value, whereas the **compute()** methods return the new value.

A very interesting use case for the **computeIfAbsent()** method is the creation of maps with lists as values. Suppose you have the following list of strings: **[one two three four five six seven].** You need to create a map, where the **keys** are the length of the words of that list, and the values are the **lists** of these words. What you need to create is the following map:



Without the **compute()** methods, you would probably write this:



Running this code produces the expected result:



By the way, you could use a **putIfAbsent()** to simplify this for loop:



But using **computeIfAbsent()** can make this code even better:



How does this code work?

If the **key** is not in the map, then the mapping function is called, which creates an **empty list**. This list is returned by the **computeIfAbsent()** method. This is the empty list in which the code adds word.

If the key is in the map, the mapping function is not called, and the current value bound to that key is returned. This is the partially filled list in which you need to add word.

This code is much more efficient than the **putIfAbsent()** one, mostly because in that case, the empty list is created only if needed. The **putIfAbsent()** call requires an existing **empty list**, which is used only if the key is not in the map. In cases where the object you add to the map has to be created on demand, then using **computeIfAbsent()** should be preferred over **putIfAbsent().**

**Merging Values:**

The **computeIfAbsent()** pattern works well if your map has values that are aggregations of other values. But there is a restriction on the structure that is supporting this **aggregation**: it has to be mutable. This is the case for **ArrayList**, and this is what the code you wrote does: it adds your values to an **ArrayList**.

Instead of creating lists of words, suppose you need to create a concatenation of words. The String class is seen here as an aggregation of other strings, but it is not a **mutable** container: you cannot use the **computeIfAbsent()** pattern to do that.

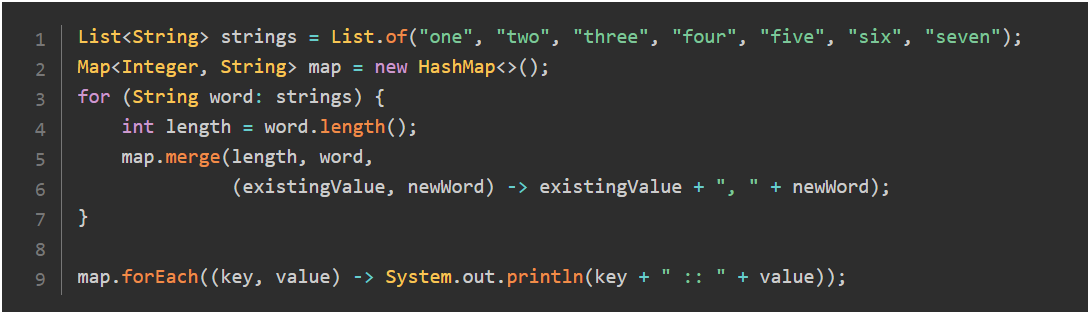
This is where the **merge()** pattern comes to the **rescue**. The **merge()** method takes three arguments:

* a key
* a value, that you need to bind to that key
* a remapping BiFunction.

If the key is not in the map or bound to a null value, then the value is bound to that key. The remapping function is not called in this case.

On the contrary, if the key is already bound to a non-null value, then the remapping function is called with the existing value, and the new value passed as an argument. If this remapping function returns null, then the key is removed from the map. The value it produces is bound to that key otherwise.

You can see this **merge()** pattern in action on the following example:



In this case, if the **length** key is not in the map, then the **merge()** call just adds it and binds it to **word**. On the other hand, if the **length** key is already in the map, then the **bifunction** is called with the existing value and **word**. The result of the bifunction then replaces the **current** value.

Running this code produces the following result:



In both patterns, **computeIfAbsent()** and **merge(),** you may be wondering why the lambda created takes an argument that is always available in the context of this lambda, and that could be captured from that context.

The answer is: **you should favor non-capturing lambdas over capturing ones, for performance reasons.**

**Keeping Keys Sorted with SortedMap and NavigableMap:**

The JDK provides two extensions of the **Map** interface: **SortedMap** and **NavigableMap**. **NavigableMap** is an extension of **SortedMap**. Both interfaces are implemented by the same class: **TreeMap**. The **TreeMap** class is a **red-black** tree, a well-known data structure.

**SortedMap** and **NavigableMap** keep their key/value pairs sorted by key. Just as for **SortedSet** and **NavigableSet**, you need to provide a way to compare these keys. You have two solutions to do this: either the class of your keys implements **Comparable**, or you provide a Comparator for your keys when creating your **TreeMap**. If you provide a **Comparator**, it will be used even if your keys are comparable.

**SortedMap** and **NavigableMap** keep their key/value pairs sorted by key. Just as for **SortedSet** and **NavigableSet**, you need to provide a way to compare these keys. You have two solutions to do this: either the **class** of your keys implements **Comparable**, or you provide a **Comparator** for your keys when creating your **TreeMap**. If you provide a **Comparator**, it will be used even if your keys are **comparable**.

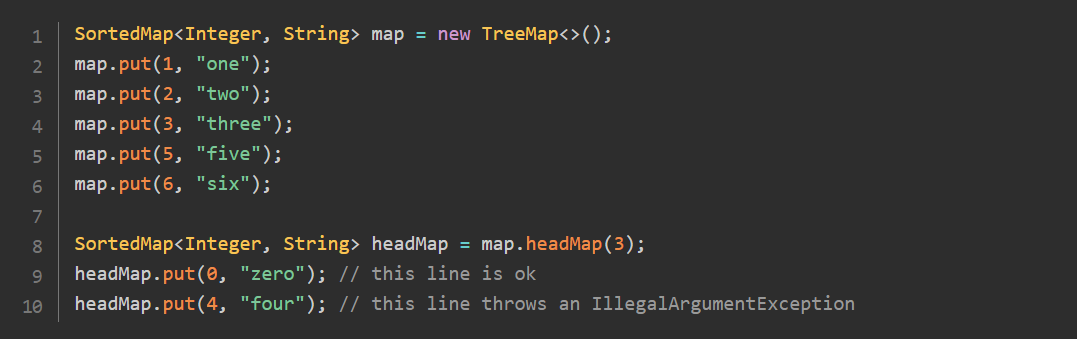
If the implementation you chose for your **SortedMap** or **NavigableMap** is **TreeMap**, then you can safely cast the set returned by a call to **keySet()** or **entrySet()** to **SortedSet** or **NavigableSet**. **NavigableMap** has a method, **navigableKeySet()** that returns an instance of **NavigableSet** that you can use instead of the plain **keySet()** method. Both methods return the same object.

The **SortedMap** interface adds the following methods to **Map**:

* **firstKey()** and **lastKey():** returns the lowest and the greatest key of your map;
* **headMap(toKey)** and **tailMap(fromKey):** returns a SortedMap whose keys are strictly less than **toKey**, or greater than or equal to **fromKey**;
* **subMap(fromKey, toKey):** returns a **SortedMap** whose keys are strictly lesser than **toKey**, or greater than or equal to **fromKey**.

These maps are instances of **SortedMap** and are views backed by this map. Any change made to this map will be seen in these views. These views can be updated, with a restriction: you cannot insert a key outside the boundaries of the map you built.

You can see this behavior on the following example:



**Accessing to Specific Keys or Entries:**

The **NavigableMap** adds more methods to **SortedMap**. The first set of methods gives you access to specific keys and entries in your map.

* **firstKey(),** **firstEntry(),** **lastEntry(),** and **lastKey():** return the lowest or greatest key or entry from this map.
* **ceilingKey(key),** **ceilingEntry(key),** **higherKey(key),** **higherEntry(key):** return the lowest key or entry greater than the provided key. The **ceiling** methods may return a key that is equal to the provided key, whereas the key returned by the **higher** methods is strictly greater.
* **floorKey(key),** **floorEntry(key),** **lowerKey(key),** **lowerEntry(key)**: return the greatest key or entry lesser than the provided key. The **floor** methods may return a key that is equal to the provided key, whereas the key returned by the **higher** methods is strictly lower.

**Accessing your Map with Queue-Like Features:**

The second set gives you queue-like features:

* **pollFirstEntry():** returns and removes the lowest entry
* **pollLastEntry():** returns and removes the greatest entry.

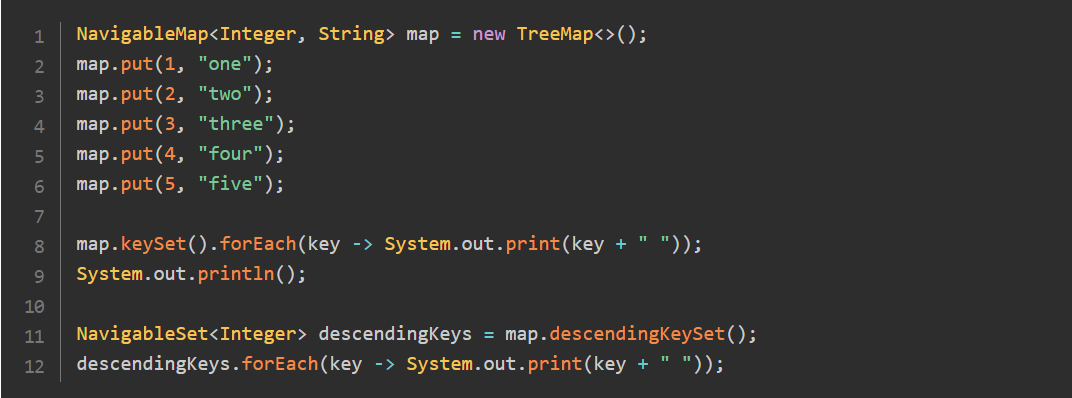
**Traversing your Map in the Reverse Order:**

The third set reverses your map, as if it had been built on the reversed comparison logic.

* **navigableKeySet()** is a convenience method that returns a **NavigableSet** so that you do not have to cast the result of **keySet()**
* **descendingKeySet():** returns a **NavigableSet** backed by the map, on which you can iterate in the descending order
* **descendingMap():** returns a **NavigableMap** with the same semantic.

Both views support element removal, but you cannot add anything through them.

Here is an example to demonstrate how you can use them.



Running this code prints out the following result.



**Getting Submap Views:**

The last set of methods give you access to views on portions of your map.

* **subMap(fromKey, fromInclusive, toKey, toInclusive)**: returns a submap where you can decide to include or not the boundaries
* **headMap(toKey, inclusive)**: same for the head map
* **tailMap(fromKey, inclusive)**: same for the tail map.

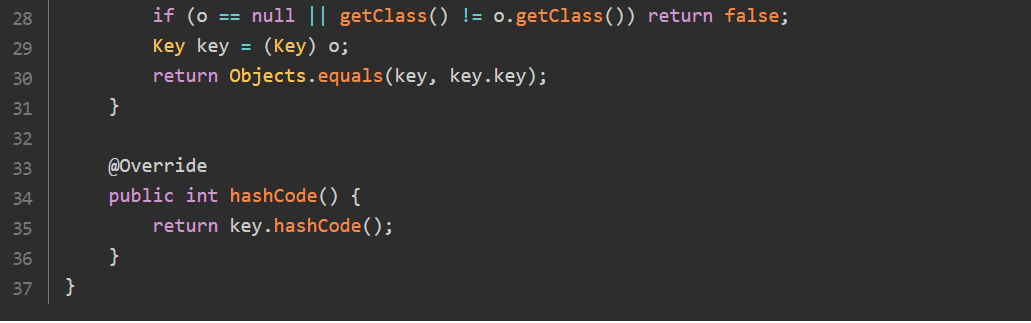
These maps are views on this map, which you can update by removing or adding key/value pairs. There is one restriction on adding elements though: you cannot add keys outside the boundaries on which the view has been created.

**Choosing Immutable Types for Your Key:**

**Avoiding the Use of Mutable Keys:**

Using mutable key is an antipattern, and you should definitely avoid doing that. The side effects you may get if you do are terrible: you may end up making the content of your map unreachable.

It is quite easy to set up an example to show that. Here is a Key class, which is just a mutable wrapper on an **String**. Note that the **equals()** and **hashCode()** methods have been overridden by a code that your IDE could generate.



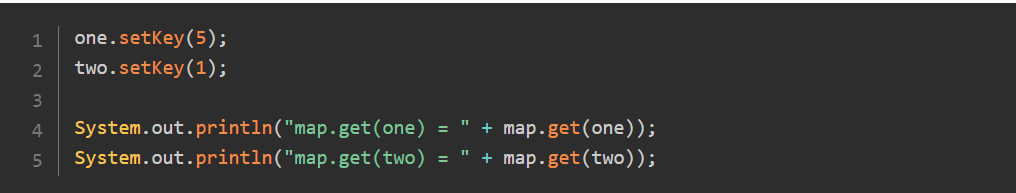
You can use this wrapper to create a map in which to put key-value pairs in.



So far this code is OK and prints out the following:



What will happen is someone mutates your key? Well, it really depends on the mutation. You can try the ones in the following example, and see what is happening when you try to get your values back.



The result is not what you want.



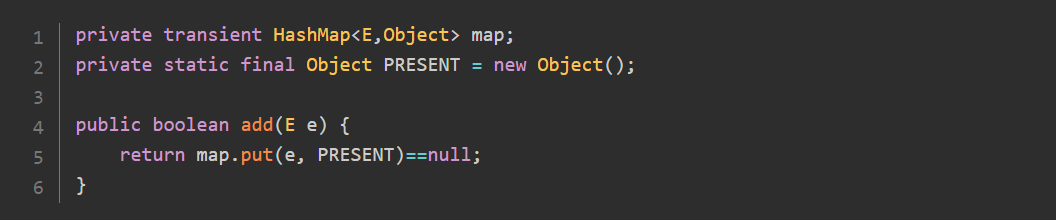
As you can see, even on a very simple example, things can go terribly wrong: the first key cannot be used to access the right value anymore, and the second one gives access to the wrong value.

In a nutshell: if you really cannot avoid using mutable keys, do not mutate them.

**Diving in the Structure of HashSet:**

You may be wondering why would it be interesting to talk about the **HashSet** class in this section? Well, it turns out that the **HashSet** class is in fact built on an internal **HashMap**. So the two classes share some common features.

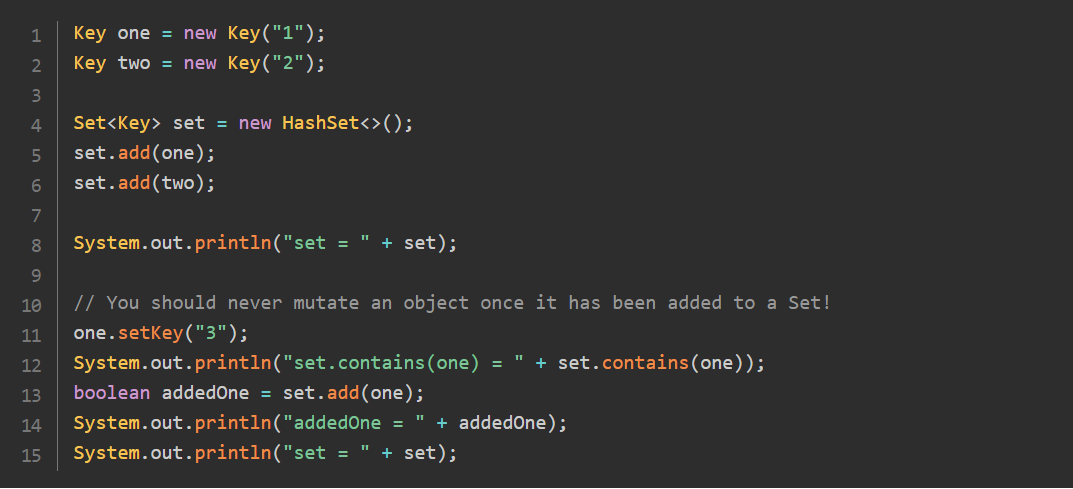
Here is the code of the **add(element)** of the HashSet class:



What you can see is that in fact, a hashset stores your object in a hashmap (the **transient** keyword is not relevant). Your objects are the keys of this hashmap, and the value is just a placeholder, an object with no significance.

The important point to remember here is that if you mutate your objects after you have added them to a set, you may come across weird bugs in your application, that will be hard to fix.

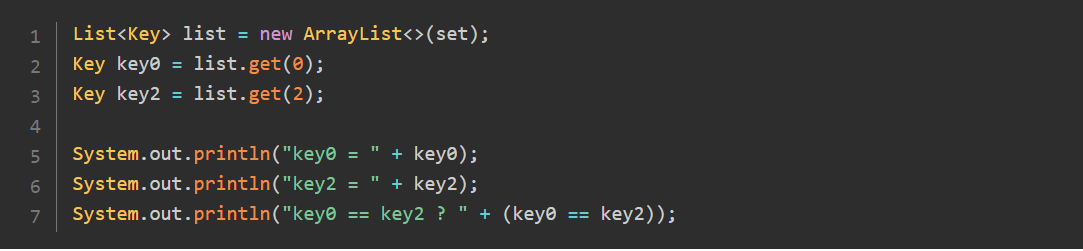
Let us take the previous example again, with the mutable **Key** class. This time, you are going to add instances of this class to a set.



Running this code produces the following result:



You can see that in fact the first element and the last element of the set are the same:



If you run this last piece of code, you will get the following result:



In this example, you saw that mutating an object once it has been added to a set can lead to having the same object more than once in this set. Simply said, do not do that!