**The Developer's Guide to Collections**

It's time for a deep dive into collections in Java, including the defining philosophy of collections, important methods, and advice for implementation.

A lot has changed since the inception of the Java programming language, but the Java Collections Framework (JCF) has been a consistent backbone of Java since its early days. Although many improvements have been made to the JCF, not the least of which has been the addition of generics and concurrency, the JCF has remained largely the same. Throughout this time, collections have become an indispensable part of nearly every Java application and learning to use these interfaces and classes has become a requirement for most aspiring Java developers.

In this series of articles, we will deep dive into the world of Java collections, starting with the general concepts of collections and the collections interface (this article), and devoting an entire article to each of the primary collection data structures: Lists, sets, and queues. By the completion of this series, we will have covered the fundamentals of each of these concepts as well as the advanced topics, including complete coverage of the primary methods of each interface and class, the intention behind each collection implementation, the benefits and trade-offs each implementation offers, and when to select a specific implementation.

It is the goal of the author that by the end of this series, the reader will have a lasting and intuitive understanding of each major collection type, and the JCF in general. To kick off this series, we will start with the story of how we got to where we are.

**A Trip Down Memory Lane**

During the early days of Java, no common collection framework existed; the best that was available were loosely collected data structures in the form of arrays, Vectors, or Hashtables. Unlike many popular programming languages at the time, Java did not combine these disparate types into a framework, with common ancestors and universal characteristics. With the introduction of the [Standard Template Library (STL)](http://www.cplusplus.com/reference/stl/) in C++ in 1994, Java lagged behind in creating a common framework with hierarchical data structures.

Up to the release of Java Development Kit (JDK) 2 in late 1998, the most popular, full-featured collections frameworks that existed were the [Generic Collection Library (JGL)](http://www4.ncsu.edu/~kaltofen/courses/Languages/JavaExamples/jgl3.1.0/doc/) by ObjectSpace and [Collections Package](http://g.oswego.edu/dl/classes/collections/) by Doug Lea. Using the JGL as a basis, Joshua Bloch (author of the [Effective Java](https://www.amazon.com/Effective-Java-3rd-Joshua-Bloch/dp/0134685997) series) designed much of the JCF that we know today. As a matter of fact, the author tags for many of the collection classes still have his name to this day. With the advent of JDK 2, a basic collections framework was introduced to Java, but this collections framework got a major upgrade in Fall of 2004 with the release of JDK 5. This Java release introduced type-erased generics, which transformed the JCF from a type-unsafe framework (requiring explicit casts when retrieving elements) to a full-fledged generic framework. JDK 5 also introduced concurrency in the JCF (through [Java Specification Request  (JSR) 166](http://g.oswego.edu/dl/concurrency-interest/)), spearheaded by the forefather of the JCF: Doug Lea.

Since that time, various upgrades have been made to the JCF, including the introduction of functional programming concepts in the form of the [Streams Application Programming Interface (API)](https://docs.oracle.com/javase/9/docs/api/java/util/stream/Stream.html), but the JCF has largely remained the same. Being that it is one of the most widely used of the Java frameworks, updates and improvements to the JCF have been one of the foremost concerns of the Java community, even from the infant days of Java.

**The Concept of a Collection**

In the Java environment, a collection is [defined as follows](https://docs.oracle.com/javase/9/docs/api/java/util/Collection.html):

**A collection represents a group of objects, known as its elements**

This definition is intentionally vague. It makes no statement about how the elements in a collection are grouped, if these elements are randomly accessible, if duplicate elements are allowed, or if elements are ordered. Instead, Java only requires that elements can be added to a collection, removed from a collection, and iterated over (without declaring an iteration order). Additionally, collections must provide queries for its current state, such as the current number of elements it contains, if the collection is empty, and if an arbitrary element is within the collection; they must also provide conversion methods to arrays (for compatibility with existing array-based applications). With the addition of the [Streams (API)](https://docs.oracle.com/javase/9/docs/api/java/util/stream/Stream.html) in JDK 8, collections must also be convertible to a stream of its elements.

Based on this description of a collection, we can define the following set of responsibilities:

* Query number of elements
* Query if collection is empty
* Query if an arbitrary element is contained in the collection
* Iterate over the elements
* Produce an array of its elements
* Add a new elements
* Remove an existing element
* Remove all existing elements
* Produce a stream of its elements

**The Collection Interface**

As expected, Java captures these responsibilities in the [Collection interface](https://docs.oracle.com/javase/9/docs/api/java/util/Collection.html), which is parameterized with a formal generic parameter, E, that represents the type of its elements. JDK 9 defines this interface as follows:

public interface Collection<E> extends Iterable<E> {

int size();

boolean isEmpty();

boolean contains(Object o);

Iterator<E> iterator();

Object[] toArray();

<T> T[] toArray(T[] a);

boolean add(E e);

boolean remove(Object o);

boolean containsAll(Collection<?> c);

boolean addAll(Collection<? extends E> c);

boolean removeAll(Collection<?> c);

default boolean removeIf(Predicate<? super E> filter) { /\* ... \*/ }

boolean retainAll(Collection<?> c);

void clear();

boolean equals(Object o);

int hashCode();

default Spliterator<E> spliterator() { /\* ... \*/ }

default Stream<E> stream() { /\* ... \*/ }

default Stream<E> parallelStream() { /\* ... \*/ }

}

**size and isEmpty**

In the vernacular of collections, the number of elements currently contained in a collection is called its **size**. Thus, in order to query the number of elements in a collection, we must call the size() method. A collection is considered **empty** (isEmpty() returns true) if its size is 0.

**contains**

The contains(Object o) method checks if the supplied object is contained in the collection according to the following rule:

**An object  a is contained in a collection if there exists at least one element b in the collection such that  a.equals(b) returns true. If a isnull, a is contained in the collection if there exists at least one nullelement in the collection.**

Stated more formally:

**An object  a is contained in a collection if there exists some elementb in the collection that satisfies the ternary expression  (a == null ? b == null : a.equals(b))**

It is important to note that this does not mean that the equals method of a will actually be invoked. It is left to the implementation to decide how to test for equality. Optimizations may be made, depending on the nature of the collection. For example, [the hashCode() method specification](https://docs.oracle.com/javase/9/docs/api/java/lang/Object.html#hashCode--)states that two objects with different hashCode() values are by definition not equal, and therefore, an implementation may favor this means of testing equality, rather than explicitly executing a.equals(b).

**iterator**

The Collection interface is actually not the top of the collections hierarchy: It extends the Iterable interface, which defines a type that can be iterated over. The JDK 9 definition for the Iterable interface is as follows:

public interface Iterable<T> {

Iterator<T> iterator();

default void forEach(Consumer<? super T> action) {

Objects.requireNonNull(action);

for (T t : this) {

action.accept(t);

}

}

default Spliterator<T> spliterator() {

return Spliterators.spliteratorUnknownSize(iterator(), 0);

}

}

The first method, iterator(), simply returns an Iterator object (which we address shortly). The forEach method is a straightforward default implementation that allows a non-null action to be performed on each of the elements in a collection. This method utilizes the for-each loop (formally called the [enhanced for loop](https://blogs.oracle.com/corejavatechtips/using-enhanced-for-loops-with-your-classes)), which is a syntactic optimization that allows a for loop to be condensed for any iterable object:

class FooIterable implements Iterable<Foo> { /\* ... \*/ }

FooIterable fooIterable;

for (Foo foo: fooIterable) {

// Do something with "foo"

}

There are some limitations to this style, such as removing an element during iteration, as we will see shortly. The last method creates a Spliterator, which can partition a collection and facilitates iteration over these partitions. Spliterators are a complex topic and are closely related with parallelism in the collections framework, and are therefore not covered in this article. The curious reader should consult the [Spliterator documentation](https://docs.oracle.com/javase/9/docs/api/java/util/Spliterator.html" \t "_blank) and [Java 8: A Quick Introduction to Parallelism and the Spliterator](https://blog.rapid7.com/2015/10/28/java-8-introduction-to-parallelism-and-spliterator/) for more information.

The main purpose of the Iterable interface is to create an Iterator object. An Iterator is the primary interface in the [Iterator pattern](https://en.wikipedia.org/wiki/Iterator_pattern) and allows for iteration over a collection. The Iterator interface is defined as follows for JDK 9:

public interface Iterator<E> {

boolean hasNext();

E next();

default void remove() {

throw new UnsupportedOperationException("remove");

}

default void forEachRemaining(Consumer<? super E> action) {

Objects.requireNonNull(action);

while (hasNext())

action.accept(next());

}

}

Following the Iterator pattern, the primary methods in this interface are the hasNext() method, which returns true if there are more elements to iterate, and the next() method, which returns the next element to be iterated and advances the current element in the iterator (i.e. ensure that the next call to next() will produce the next element of the collection, not the same element *ad infinitum*). Beyond these fundamental methods, the Iterator interface also includes two important methods: remove() and forEachRemaining(Consumer<? super E> action).

The remove()) method is essential to removing elements from a collection during iteration. In general, it is not permissible to iterate over a collection using an enhanced for loop and remove an element from the collection in the body of the for loop. This results in a [ConcurrentModificationException](https://docs.oracle.com/javase/9/docs/api/java/util/ConcurrentModificationException.html" \t "_blank) being thrown. For example, the following results in a ConcurrentModificationException:

for (Foo foo: fooIterable) {

fooIterable.remove(foo);

}

Instead, remove() must be called on an Iterator object to remove an element from a collection while iterating over that collection. For example:

for(Iterator<Foo> i = fooIterable.iterator(); i.hasNext();) {

Foo foo = i.next();

i.remove();

}

Note that next() must be called before calling remove(), as the next() method advances the current element in the iterator. While the combination of the next(), hasNext(), and remove() method cover a vast majority of the functionality that a developer will commonly use when dealing with iterators, there are countless great resources that go much deeper on this important topic. For more information, consult the [Iterator documentation](https://docs.oracle.com/javase/9/docs/api/java/util/Iterator.html) and the [Oracle Collection Interface documentation](https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html#Iterator).

**toArray**

Due to the limitation of Java generics, conversions of collections to arrays is quirky and two methods are provided. The first is the simple toArray() method that returns an array of Objects that holds the elements of the collection ordered by the same rules that govern the order of the elements obtained through an iterator associated with the collection (i.e. if there are ordering rules established for an iterator returned by iterator(), these rules also govern the order of the elements in this array). This method is a non-generic method and the type of the elements in the array *do not* reflect the type of the elements, as specified by the formal generic parameter for the collection.

The second method is the toArray(T[] a) method, which returns an array of the elements in the collection, but retains the type of the elements. Thus, the array returned by this method is an array of objects with the same type as the formal generic parameter of the collection. Due [type erasure](https://docs.oracle.com/javase/tutorial/java/generics/erasure.html) of generics in Java, the type of the formal generic parameter is inaccessible at run-time, and therefore, creating an array at run-time with the same type as the elements is not feasible. Thus, the caller is responsible for providing the type of array at run-time (in the form of an array of the element type). If the provided array has a size equal to greater than the size of the collection, the supplied array is filled with the elements of the collection and the element directly following the last element of the collection is set to null (if the length of the array minus the size of the collection is greater than or equal to 1). If the length of the supplied array is less than the size of the collection, a new array is returned with a length that matches the size of the collection and the type of the supplied array.

For example, a call to this parameterized method would resemble the following:

Collection<Foo> foos = someMethod();

Foo[] fooArray = foos.toArray(new Foo[0]);

It can be tempting to optimize this call by pre-allocating an array of the same length as the size of the collection. For example:

Collection<Foo> foos = someMethod();

Foo[] fooArray = foos.toArray(new Foo[foos.size()]);

As noted by Joshua Bloch in Effective Java, 3rd Edition (pg. 248), Item 55, this pre-allocation optimization should be avoided. For a quantitative analysis of the performance differences between these two array pre-allocation techniques, see [Arrays of Wisdom of the Ancients](https://shipilev.net/blog/2016/arrays-wisdom-ancients/).

**add**

The add(E e) method adds an element to a collection and returns true if the collection was changed. Implementations are left to decide if e is acceptable. For example, some implementations may not accept duplicate values (i.e. if contains(e) is true) while others may not accept null values.

**remove**

The remove(Object o) method removes one element from a collection if at least one element equal to the supplied object is contained in the collection. If an element is removed, this method returns true. The equality rules for removal are identical to those of the contains(Object o) method.

**containsAll, addAll, and removeAll**

The containsAll(Collection<?> c) method returns true if and only if all of the elements of c are contained in the collection. Likewise, the addAll(Collection<?> c) method adds all of the elements in c to the collection, returning true if the collection is altered (i.e. at least one addition was performed). Note that the behavior of the addAll method is undefined if the collection is modified between the initiation of the addAll method and its completion. Lastly, removeAll(Collection<?>c) removes all elements in common with c, returning true if the collection was modified (i.e. if at least one element was removed). Upon completion, the collection is guaranteed to contain common elements with c.

**removeIf**

The removeIf(Predicate<? super E> filter) default implementation removes all elements that satisfy the supplied predicate. In practice, this method filters out any element that satisfies the supplied predicate and returns true if the collection was modified (i.e. at least one element was removed). The implementation of this method is as follows:

default boolean removeIf(Predicate<? super E> filter) {

Objects.requireNonNull(filter);

boolean removed = false;

final Iterator<E> each = iterator();

while (each.hasNext()) {

if (filter.test(each.next())) {

each.remove();

removed = true;

}

}

return removed;

}

Internally, this method uses the iterator provided by the collection to test each of the elements in the collection, removing those that do not satisfy the supplied predicate (as demonstrated in the iterator section above).

**retainAll**

The retainAll(Collection<?> c) method removes all elements of the collection not in common with c. This amounts to the intersection of the collection with c. For example, if a collection contains the elements [1, 2, 2, 3, 4, 5, 5], and a collection containing [1, 2, 4, 6] is supplied to the retainAll method, the original collection will be reduced to [1, 2, 2, 4]. A value of true is returned if this method modifies the collection (i.e. if at least one element is removed).

**clear**

Removes all elements from the collection. Upon completion of this method, the collection is considered empty.

**equals and hashCode**

The equals(Object o) and hashCode() methods mirror those of all subclasses of the Object class in Java. Implementations of collections who customize the equals and hashCode methods are obligated to follow the general restrictions placed on all implementations of the Object class, found in the [Object class documentation](https://docs.oracle.com/javase/9/docs/api/java/lang/Object.html).

**spliterator**

The Collection interface overrides the default spliterator implementation provided by the Iterable interface with its own, replacing a Spliterator associated with the Iterator returned by the Iterable interface with one associated with the collection itself. The default implementation of this method is as follows:

default Spliterator<E> spliterator() {

return Spliterators.spliterator(this, 0);

}

For more information on Spliterators, see the [Spliterator documentation](https://docs.oracle.com/javase/9/docs/api/java/util/Spliterator.html" \t "_blank).

**stream and parallelStream**

One of the major additions to JDK 8 was the inclusion of the [Streams API](https://docs.oracle.com/javase/9/docs/api/java/util/stream/Stream.html). This API introduces functional programming semantics to Java collections, treating collections as a stream of data that can be mapped, filtered, reduced, etc. The default implementations of the stream-based methods for the Collection interface are as follows:

default Stream<E> stream() {

return StreamSupport.stream(spliterator(), false);

}

default Stream<E> parallelStream() {

return StreamSupport.stream(spliterator(), true);

}

As with Spliterators, streams are a very involved topic and beyond the scope of this series. The curious reader can find more information at the [Stream class documentation](https://docs.oracle.com/javase/9/docs/api/java/util/stream/Stream.html).

**Implicit Rules**

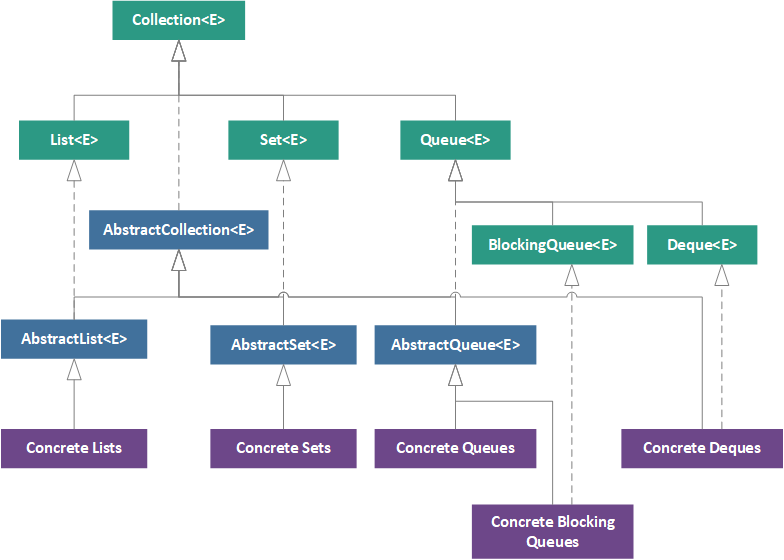
While most of the rules regarding collection implementations are checked at compile-time with language constructs, some rules are extra-linguistic. Although not checked at compile-time, *all* implementations of collections should abide by the following rules:

* Have a no-args constructor that creates an empty collection (of the implementation type) and a constructor that accepts a Collection and creates a copy of the supplied Collection (**conversion constructor** or **copy constructor**)
* Unsupported destructive methods (i.e. methods that modify the collection) should throw an [UnsupportedOperationException](https://docs.oracle.com/javase/9/docs/api/java/lang/UnsupportedOperationException.html" \t "_blank) if not supported; for example, calling add or remove on an immutable collection should result in an [UnsupportedOperationException](https://docs.oracle.com/javase/9/docs/api/java/lang/UnsupportedOperationException.html" \t "_blank) being thrown
* Synchronization is determined by each implementation

**Collection Hierarchy**

The power and utility of the JCF does not come from a single Collection interface, but rather, the various other interfaces, abstract classes, and concrete implementations that make up the framework. Of these other collection types, three stand out in their ascendency: (1) List, (2) Set, and (3) Queue. A list is an ordered collection, or sequence of elements; a set is a collection that does not allow duplicates (and allows at most one null element), mirroring a mathematical set; and a queue is a collection designed for processing and usually orders its elements in some front or back ordering, such as First-in-First-out (FIFO), Last-in-First-out (LIFO), or by natural ordering established through a comparator (as with priority queues).

As we will see in the following articles in this series, these three concepts cover most of the use cases for collections. This hierarchy is illustrated in the figure below, with green boxes representing interfaces, blue boxes representing abstract classes, and purple boxes representing groups of concrete implementations. Note that an excellent interactive depiction of the collections hierarchy can be found on [Falkhausen](http://www.falkhausen.de/Java-9/java.util/Collection-Hierarchy.html" \t "_blank).



It is important to note that in general, the collections framework follows a specific pattern for its implementations, where each concrete implementation of the Collection interface inherits from an abstract class closely related to its type (i.e. concrete list classes extend the AbstractList abstract class), which in turn extends the AbstractCollection abstract class and an interface closely associated with the collection type (i.e. the AbstractList abstract class extends the AbstractCollection abstract class and implements the List interface). This closely associated interface then extends the Collection interface, ensuring that all of the features associated with the collections, in general, are associated with the concrete collection type (i.e. the List interface extends the Collection interface).

As we will see in future articles, each of these collection types introduces their own constraints and restrictions that allow them to more precisely define the behavior of a collection. For example, due to the ordered nature of lists, the List interface includes a method for random access. We will cover this feature, and all of the other idiosyncrasies of lists in the next article in this series.