forEach to the rescue.

* If we take the Java Collections framework then Collection<E> is the root interface.
* Now, Collection<E> interface extends Iterable<E> interface.
* And as of Java 8 Iterable<E> interface is enhanced and provided more functionality. This means those functionality can be directly used by any Collection in entire Collection Framework due to hierarchy.
* Now, in previous post we saw that there are few problems that we might suffer from for using the iteration methods. We can use the default method **forEach(Consumer<? super T> action)** for our purpose.
* forEach method has parameter of type *Consumer*. Consumer is Functional Interface. Let us understand forEach method first then proceed further.

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

Objects.*requireNonNull*(action);

**for** (T t : **this**) {

action.accept(t);

}

}

As of Java 8 we can write method definition in interface and mark it default. Read the tutorial for default methods here.

As per [javadoc](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html#forEach-java.util.function.Consumer-), this method performs the given action for each element until all elements are processed or action throws an exception.

So let us write a forEach(..) method with a Consumer as anonymous inner class syntax.

/\*\*

\* languages is list with following values.

\* [Java, JavaScript, C#, Python]

\* \*/

languages.forEach(**new** Consumer<String>() {

**public** **void** accept(**final** String language) {

System.***out***.println(language);

}

});

Yes this method does the following.

1. Accepts the parameter String language.
2. Prints it.
3. Repeat till list is empty

Here, accept method just prints the elements of the list. We can do much more but that’s for later.

But, no one likes the Anonymous inner class as the syntax is much clumsier and method looks verbose.

1. GOOD NEWS, you can use lambda operator here. Why? Because Consumer is Functional Interface.

languages.forEach((**final** String language) -> System.***out***.println(language));

1. We can drop the type of the parameter as we have only one method in Functional Interface on which we can apply lambda operator. So it will look like this.

languages.forEach((language) -> System.***out***.println(language));

1. If there is single parameter then we can eliminate the brackets outside the parameter too.

languages.forEach(language -> System.***out***.println(language));

1. Excellent, now let us make this more concise by using method reference. We will learn about this syntax later.

languages.forEach(System.***out***::println);

forEach method solves the problems that we faced in other iteration methods. We will see how to use forEach in transforming list and other operations.