**Collections in Java**

Any group of individual objects which are represented as a single unit is known as the collection of the objects. In Java, a separate framework named the *“Collection Framework”* has been defined in JDK 1.2 which holds all the collection classes and interface in it.

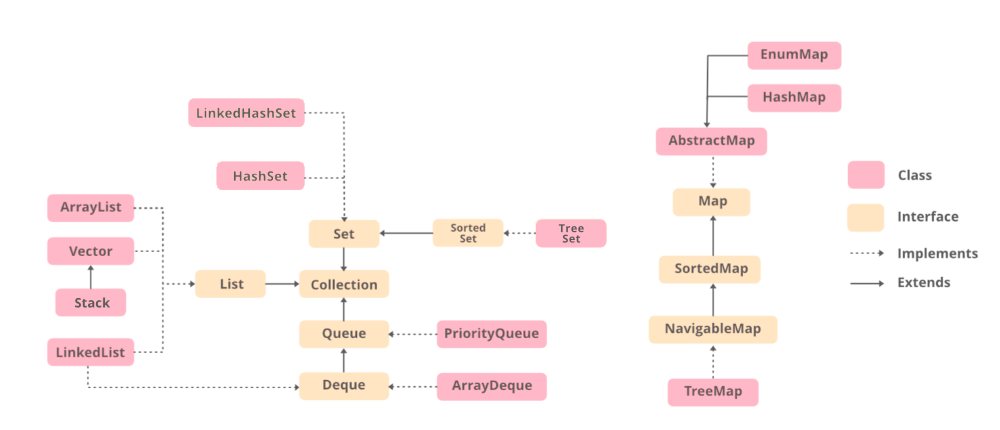
The Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are the two main “root” interfaces of Java collection classes.

**What is a Framework?**

A framework is a set of [classes](https://www.geeksforgeeks.org/classes-objects-java/) and [interfaces](https://www.geeksforgeeks.org/interfaces-in-java/) which provide a ready-made architecture. In order to implement a new feature or a class, there is no need to define a framework. However, an optimal object-oriented design always includes a framework with a collection of classes such that all the classes perform the same kind of task.

**Hierarchy of the Collection Framework**

The utility package, (java.util) contains all the classes and interfaces that are required by the collection framework. The collection framework contains an interface named an iterable interface which provides the iterator to iterate through all the collections. This interface is extended by the main collection interface which acts as a root for the collection framework. All the collections extend this collection interface thereby extending the properties of the iterator and the methods of this interface. The following figure illustrates the hierarchy of the collection framework.



Before understanding the different components in the above framework, let’s first understand a class and an interface.

* [**Class**](https://www.geeksforgeeks.org/classes-objects-java/)**:** A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type.
* [**Interface**](https://www.geeksforgeeks.org/interfaces-in-java/)**:** Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body). Interfaces specify what a class must do and not how. It is the blueprint of the class.

**Methods of the Collection Interface**

This interface contains various methods which can be directly used by all the collections which implement this interface. They are:

| Method | Description |
| --- | --- |
| [**add(Object)**](https://www.geeksforgeeks.org/collection-add-method-in-java-with-examples/) | This method is used to add an object to the collection. |
| [**addAll(Collection c)**](https://www.geeksforgeeks.org/collections-addall-method-in-java-with-examples/) | This method adds all the elements in the given collection to this collection. |
| [**clear()**](https://www.geeksforgeeks.org/collection-clear-method-in-java-with-examples/) | This method removes all of the elements from this collection. |
| [**contains(Object o)**](https://www.geeksforgeeks.org/collection-contains-method-in-java-with-examples/) | This method returns true if the collection contains the specified element. |
| **containsAll(Collection c)** | This method returns true if the collection contains all of the elements in the given collection. |
| **equals(Object o)** | This method compares the specified object with this collection for equality. |
| **hashCode()** | This method is used to return the hash code value for this collection. |
| [**isEmpty()**](https://www.geeksforgeeks.org/collection-isempty-method-in-java-with-examples/) | This method returns true if this collection contains no elements. |
| **iterator()** | This method returns an iterator over the elements in this collection. |
| [**max()**](https://www.geeksforgeeks.org/collections-max-method-in-java-with-examples/) | This method is used to return the maximum value present in the collection. |
| **parallelStream()** | This method returns a parallel Stream with this collection as its source. |
| **remove(Object o)** | This method is used to remove the given object from the collection. If there are duplicate values, then this method removes the first occurrence of the object. |
| **removeAll(Collection c)** | This method is used to remove all the objects mentioned in the given collection from the collection. |
| **removeIf(Predicate filter)** | This method is used to remove all the elements of this collection that satisfy the given [predicate](https://www.geeksforgeeks.org/mathematic-logic-predicates-quantifiers/). |
| **retainAll(Collection c)** | This method is used to retain only the elements in this collection that are contained in the specified collection. |
| **size()** | This method is used to return the number of elements in the collection. |
| **spliterator()** | This method is used to create a [Spliterator](https://www.geeksforgeeks.org/java-program-to-convert-iterator-to-spliterator/) over the elements in this collection. |
| **stream()** | This method is used to return a sequential Stream with this collection as its source. |
| **toArray()** | This method is used to return an array containing all of the elements in this collection. |

**Interfaces that extend the Collections Interface**

The collection framework contains multiple interfaces where every interface is used to store a specific type of data. The following are the interfaces present in the framework.

**1. Iterable Interface:** This is the root interface for the entire collection framework. The collection interface extends the iterable interface. Therefore, inherently, all the interfaces and classes implement this interface. The main functionality of this interface is to provide an iterator for the collections. Therefore, this interface contains only one abstract method which is the iterator. It returns the 

*Iterator iterator();*

**2. Collection Interface:** This interface extends the iterable interface and is implemented by all the classes in the collection framework. This interface contains all the basic methods which every collection has like adding the data into the collection, removing the data, clearing the data, etc. All these methods are implemented in this interface because these methods are implemented by all the classes irrespective of their style of implementation. And also, having these methods in this interface ensures that the names of the methods are universal for all the collections. Therefore, in short, we can say that this interface builds a foundation on which the collection classes are implemented.

**3.**[**List Interface:**](https://www.geeksforgeeks.org/list-interface-java-examples/) This is a child interface of the collection interface. This interface is dedicated to the data of the list type in which we can store all the ordered collection of the objects. This also allows duplicate data to be present in it. This list interface is implemented by various classes like [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/), [Vector](https://www.geeksforgeeks.org/java-util-vector-class-java/), [Stack](https://www.geeksforgeeks.org/stack-class-in-java/), etc. Since all the subclasses implement the list, we can instantiate a list object with any of these classes. For example, 

*List <T> al = new ArrayList<> ();   
List <T> ll = new LinkedList<> ();   
List <T> v = new Vector<> ();*

*Where T is the type of the object*

***The classes which implement the List interface are as follows:***

1. [**ArrayList:**](https://www.geeksforgeeks.org/arraylist-in-java/) ArrayList provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed. The size of an ArrayList is increased automatically if the collection grows or shrinks if the objects are removed from the collection. Java ArrayList allows us to randomly access the list. ArrayList can not be used for [primitive types](https://www.geeksforgeeks.org/primitive-data-type-vs-object-data-type-in-java-with-examples/), like int, char, etc. We will need a [wrapper class](https://www.geeksforgeeks.org/wrapper-classes-java/) for such cases. Let’s understand the ArrayList with the following example:

IDE

**B.**[**LinkedList:**](https://www.geeksforgeeks.org/linked-list-in-java/) LinkedList class is an implementation of the [LinkedList data structure](https://www.geeksforgeeks.org/data-structures/linked-list/) which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Let’s understand the LinkedList with the following example: IDE\*\*

**C.**[**Vector:**](https://www.geeksforgeeks.org/java-util-vector-class-java/) A vector provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed. This is identical to ArrayList in terms of implementation. However, the primary difference between a vector and an ArrayList is that a Vector is synchronized and an ArrayList is non-synchronized. Let’s understand the Vector with an example:IDE\*\*

**D.**[**Stack**](https://www.geeksforgeeks.org/stack-class-in-java/)**:** Stack class models and implements the [Stack data structure](https://www.geeksforgeeks.org/stack-data-structure/). The class is based on the basic principle of *last-in-first-out*. In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek. The class can also be referred to as the subclass of Vector. Let’s understand the stack with an example:IDE\*\*

**Note:**Stack is a subclass of Vector and a legacy class. It is thread-safe which might be overhead in an environment where thread safety is not needed. An alternate to Stack is to use [ArrayDequeue](https://www.geeksforgeeks.org/arraydeque-in-java/) which is not thread-safe and has faster array implementation.

**4.**[**Queue Interface**](https://www.geeksforgeeks.org/queue-interface-java/)**:** As the name suggests, a queue interface maintains the FIFO(First In First Out) order similar to a real-world queue line. This interface is dedicated to storing all the elements where the order of the elements matter. For example, whenever we try to book a ticket, the tickets are sold on a first come first serve basis. Therefore, the person whose request arrives first into the queue gets the ticket. There are various classes like [PriorityQueue](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/), [ArrayDeque](https://www.geeksforgeeks.org/arraydeque-in-java/), etc. Since all these subclasses implement the queue, we can instantiate a queue object with any of these classes. For example, 

*Queue <T> pq = new PriorityQueue<> ();   
Queue <T> ad = new ArrayDeque<> ();*

*Where T is the type of the object.*

***The most frequently used implementation of the queue interface is the PriorityQueue.***  
   
[**Priority Queue:**](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/) A PriorityQueue is used when the objects are supposed to be processed based on the priority. It is known that a queue follows the First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority and this class is used in these cases. The PriorityQueue is based on the priority heap. The elements of the priority queue are ordered according to the natural ordering, or by a [Comparator](https://www.geeksforgeeks.org/comparator-interface-java/) provided at queue construction time, depending on which constructor is used. Let’s understand the priority queue with an example:IDE\*\*

**5.**[**Deque Interface**](https://www.geeksforgeeks.org/deque-interface-java-example/)**:** This is a very slight variation of the [queue data structure](https://www.geeksforgeeks.org/queue-data-structure/). [Deque](https://www.geeksforgeeks.org/deque-set-1-introduction-applications/), also known as a double-ended queue, is a data structure where we can add and remove the elements from both ends of the queue. This interface extends the queue interface. The class which implements this interface is [ArrayDeque](https://www.geeksforgeeks.org/arraydeque-in-java/). Since ArrayDeque class implements the Deque interface, we can instantiate a deque object with this class. For example, 

*Deque<T> ad = new ArrayDeque<> ();*

*Where T is the type of the object.*

***The class which implements the deque interface is ArrayDeque.***

[**ArrayDeque:**](https://www.geeksforgeeks.org/arraydeque-in-java/) ArrayDeque class which is implemented in the collection framework provides us with a way to apply resizable-array. This is a special kind of array that grows and allows users to add or remove an element from both sides of the queue. Array deques have no capacity restrictions and they grow as necessary to support usage. Let’s understand ArrayDeque with an example:

IDE\*\*

**6.**[**Set Interface**](https://www.geeksforgeeks.org/set-in-java/)**:** A set is an unordered collection of objects in which duplicate values cannot be stored. This collection is used when we wish to avoid the duplication of the objects and wish to store only the unique objects. This set interface is implemented by various classes like [HashSet](https://www.geeksforgeeks.org/hashset-in-java/), [TreeSet](https://www.geeksforgeeks.org/treeset-in-java-with-examples/), [LinkedHashSet](https://www.geeksforgeeks.org/linkedhashset-in-java-with-examples/), etc. Since all the subclasses implement the set, we can instantiate a set object with any of these classes. For example,

*Set<T> hs = new HashSet<> ();   
Set<T> lhs = new LinkedHashSet<> ();   
Set<T> ts = new TreeSet<> ();*

*Where T is the type of the object.*

***The following are the classes that implement the Set interface:***

1. [**HashSet:**](https://www.geeksforgeeks.org/hashset-in-java/) The HashSet class is an inherent implementation of the hash table data structure. The objects that we insert into the HashSet do not guarantee to be inserted in the same order. The objects are inserted based on their hashcode. This class also allows the insertion of NULL elements. Let’s understand HashSet with an example:IDE\*\*

1. [**LinkedHashSet**](https://www.geeksforgeeks.org/linkedhashset-in-java-with-examples/)**:** A LinkedHashSet is very similar to a HashSet. The difference is that this uses a doubly linked list to store the data and retains the ordering of the elements. Let’s understand the LinkedHashSet with an example: IDE\*\*

**7.**[**Sorted Set Interface**](https://www.geeksforgeeks.org/sortedset-java-examples/)**:** This interface is very similar to the set interface. The only difference is that this interface has extra methods that maintain the ordering of the elements. The sorted set interface extends the set interface and is used to handle the data which needs to be sorted. The class which implements this interface is TreeSet. Since this class implements the SortedSet, we can instantiate a SortedSet object with this class. For example,

*SortedSet<T> ts = new TreeSet<> ();*

*Where T is the type of the object.*

***The class which implements the sorted set interface is TreeSet.***  
   
[**TreeSet:**](https://www.geeksforgeeks.org/treeset-in-java-with-examples/) The TreeSet class uses a Tree for storage. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. This must be consistent with equals if it is to correctly implement the Set interface. It can also be ordered by a Comparator provided at set creation time, depending on which constructor is used. Let’s understand TreeSet with an example:IDE\*\*

**8.**[**Map Interface**](https://www.geeksforgeeks.org/map-interface-java-examples/)**:** A map is a data structure that supports the key-value pair mapping for the data. This interface doesn’t support duplicate keys because the same key cannot have multiple mappings. A map is useful if there is data and we wish to perform operations on the basis of the key. This map interface is implemented by various classes like [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/), [TreeMap](https://www.geeksforgeeks.org/treemap-in-java/), etc. Since all the subclasses implement the map, we can instantiate a map object with any of these classes. For example,

*Map<T> hm = new HashMap<> ();   
Map<T> tm = new TreeMap<> ();  
   
Where T is the type of the object.*

***The frequently used implementation of a Map interface is a HashMap.***  
   
[**HashMap**](https://www.geeksforgeeks.org/java-util-hashmap-in-java-with-examples/)**:** HashMap provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs. To access a value in a HashMap, we must know its key. HashMap uses a technique called Hashing. Hashing is a technique of converting a large String to a small String that represents the same String so that the indexing and search operations are faster. HashSet also uses HashMap internally. Let’s understand the HashMap with an example:IDE\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*End\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

File Handling

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

* **InPutStream** − The InputStream is used to read data from a source.
* **OutPutStream** − The OutputStream is used for writing data to a destination.



Java provides strong but flexible support for I/O related to files and ne

works but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one −

### **Byte Streams**

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file −

**Example**

import java.io.\*;

public class CopyFile {

public static void main(String args[]) throws IOException {

FileInputStream in = null;

FileOutputStream out = null;

try {

in = new FileInputStream("input.txt");

out = new FileOutputStream("output.txt");

int c;

while ((c = in.read()) != -1) {

out.write(c);

}

}finally {

if (in != null) {

in.close();

}

if (out != null) {

out.close();

}

}

}

}

Now let's have a file **input.txt** with the following content −

This is test for copy file.

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

$javac CopyFile.java

$java CopyFile

# **Java FileInputStream Class**

Java FileInputStream class obtains input bytes from a [file](https://www.javatpoint.com/java-file-class). It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data. But, for reading streams of characters, it is recommended to use FileReader class.

## Java FileInputStream class declaration

Let's see the declaration for java.io.FileInputStream class:

1. **public** **class** FileInputStream **extends** InputStream
2. **Java Copy File – Stream**

This is the conventional way of file copy in java. Here we create two Files – source and destination. Then we create [InputStream](https://www.journaldev.com/19187/java-fileinputstream) from source and write it to the destination file using [OutputStream](https://www.journaldev.com/19438/java-fileoutputstream) for java copy file operation.

Here is the method that can be used for java copy file using streams.

private static void copyFileUsingStream(File source, File dest) throws IOException {

InputStream is = null;

OutputStream os = null;

try {

is = new FileInputStream(source);

os = new FileOutputStream(dest);

byte[] buffer = new byte[1024];

int length;

while ((length = is.read(buffer)) > 0) {

os.write(buffer, 0, length);

}

} finally {

is.close();

os.close();

}

}

Reader & Writer:

The [**Java Reader**](https://jenkov.com/tutorials/java-io/reader.html) (java.io.Reader) and [**Java Writer**](https://jenkov.com/tutorials/java-io/writer.html) class (java.io.Writer) in Java IO work much like the [**InputStream**](https://jenkov.com/tutorials/java-io/inputstream.html) and [**OutputStream**](https://jenkov.com/tutorials/java-io/outputstream.html) with the exception that Reader and Writer are character based. They are intended for reading and writing text. The InputStream and OutputStream are byte based.

## Reader

The Java Reader is the base class of all Reader's in the Java IO API. Subclasses include a BufferedReader, PushbackReader, InputStreamReader, StringReader and several others.

Here is a simple Java IO Reader example:

Reader reader = new FileReader("c:\\data\\myfile.txt");

int data = reader.read();//listen()

while(data != -1){

char dataChar = (char) data;

data = reader.read();

}

## Writer

The Java Writer class is the base class of all Writers in the Java IO API. Subclasses include BufferedWriter and PrintWriter among others.

Here is a simple Java IO Writer example:

Writer writer = new FileWriter("c:\\data\\file-output.txt");

writer.write("Hello World Writer");

writer.close();

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*End\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Lambda expressions:

Lambda expressions basically express instances of functional interfaces (An interface with single abstract method is called functional interface. An example is java.lang.Runnable). lambda expressions implement the only abstract function and therefore implement functional interfaces

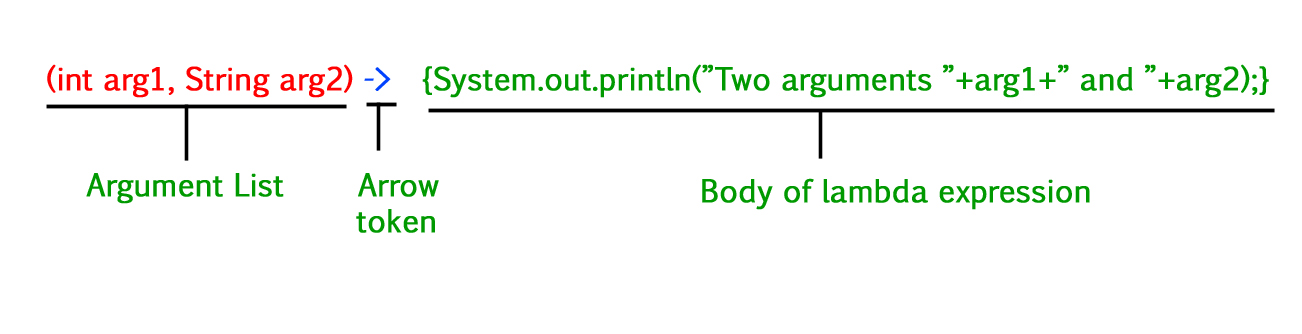
lambda expressions are added in Java 8 and provide below functionalities.

* Enable to treat functionality as a method argument, or code as data.
* A function that can be created without belonging to any class.
* A lambda expression can be passed around as if it was an object and executed on demand.

|  |
| --- |
| // Java program to demonstrate lambda expressions  // to implement a user defined functional interface.    // A sample functional interface (An interface with  // single abstract method  **interface** FuncInterface  {      // An abstract function  **void** abstractFun(**int** x);        // A non-abstract (or default) function  **default** **void** normalFun()      {         System.out.println("Hello");      }  }    **class** Test  {  **public** **static** **void** main(String args[])      {          // lambda expression to implement above          // functional interface. This interface          // by default implements abstractFun()          FuncInterface fobj = (**int** x)->System.out.println(2\*x);            // This calls above lambda expression and prints 10.          fobj.abstractFun(5);      }  } |

Output:

10

[](https://media.geeksforgeeks.org/wp-content/uploads/lambda-expression.jpg)  
**Syntax:**

lambda operator -> body

where lambda operator can be:

* **Zero parameter:**

() -> System.out.println("Zero parameter lambda");

* **One parameter:**–

(p) -> System.out.println("One parameter: " + p);

It is not mandatory to use parentheses, if the type of that variable can be inferred from the context

* **Multiple parameters :**

(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);

Please note: Lambda expressions are just like functions and they accept parameters just like functions.

|  |
| --- |
| // A Java program to demonstrate simple lambda expressions  **import** java.util.ArrayList;  **class** Test  {  **public** **static** **void** main(String args[])      {          // Creating an ArrayList with elements          // {1, 2, 3, 4}          ArrayList<Integer> arrL = **new** ArrayList<Integer>();          arrL.add(1);          arrL.add(2);          arrL.add(3);          arrL.add(4);    for(){  syso(arr[i])  }          // Using lambda expression to print all elements          // of arrL          arrL.forEach(n -> System.out.println(n));            // Using lambda expression to print even elements          // of arrL          arrL.forEach(n -> { **if** (n%2 == 0) System.out.println(n); });      }  } |

Output :

1

2

3

4

2

4

Note that lambda expressions can only be used to implement functional interfaces. In the above example also, the lambda expression implements [Consumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) Functional Interface.

**Streams:**

Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.  
The features of Java stream are –

* A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Different Operations On Streams-  
**Intermediate Operations:**

1. **map:**The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.  
   List number = Arrays.asList(2,3,4,5);  
   List square = number.stream().map(x->x\*x).collect(Collectors.toList());
2. **filter:** The filter method is used to select elements as per the Predicate passed as argument.  
   List names = Arrays.asList("Reflection","Collection","Stream");  
   List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
3. **sorted:** The sorted method is used to sort the stream.  
   List names = Arrays.asList("Reflection","Collection","Stream");  
   List result = names.stream().sorted().collect(Collectors.toList());

**Terminal Operations:**

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.  
   List number = Arrays.asList(2,3,4,5,3);  
   Set square = number.stream().map(x->x\*x).collect(Collectors.toSet());
2. **forEach:** The forEach method is used to iterate through every element of the stream.  
   List number = Arrays.asList(2,3,4,5);  
   number.stream().map(x->x\*x).forEach(y->System.out.println(y));
3. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.  
   The reduce method takes a BinaryOperator as a parameter.

List number = Arrays.asList(2,3,4,5);  
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

Here ans variable is assigned 0 as the initial value and i is added to it .

**Program to demonstrate the use of Stream**

|  |
| --- |
| //a simple program to demonstrate the use of stream in java  **import** java.util.\*;  **import** java.util.stream.\*;    **class** Demo  {  **public** **static** **void** main(String args[])    {        // create a list of integers      List<Integer> number = Arrays.asList(2,3,4,5);        // demonstration of map method      List<Integer> square = number.stream().map(x -> x\*x).                             collect(Collectors.toList());      System.out.println(square);        // create a list of String      List<String> names =                  Arrays.asList("Reflection","Collection","Stream");        // demonstration of filter method      List<String> result = names.stream().filter(s->s.startsWith("S")).                            collect(Collectors.toList());      System.out.println(result);        // demonstration of sorted method      List<String> show =              names.stream().sorted().collect(Collectors.toList());      System.out.println(show);        // create a list of integers      List<Integer> numbers = Arrays.asList(2,3,4,5,2);        // collect method returns a set      Set<Integer> squareSet =           numbers.stream().map(x->x\*x).collect(Collectors.toSet());      System.out.println(squareSet);        // demonstration of forEach method      number.stream().map(x->x\*x).forEach(y->System.out.println(y));        // demonstration of reduce method  **int** even =         number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);        System.out.println(even);    }  } |

Output:

[4, 9, 16, 25]

[Stream]

[Collection, Reflection, Stream]

[16, 4, 9, 25]

4

9

16

25

6

**Filter:**

## Stream.filter()

The filter() method is an intermediate operation of the Stream interface that allows us to filter elements of a stream that match a given Predicate:

Stream<T> **filter**(Predicate<? super T> predicate)

To see how this works, let's create a Customer class:

**public** **class** **Customer** {

**private** String name;

**private** **int** points;

//Constructor and standard getters

}

In addition, let's create a collection of customers:

**Customer** john = **new** **Customer**("John P.", 15);

**Customer** sarah = **new** **Customer**("Sarah M.", 200);

**Customer** charles = **new** **Customer**("Charles B.", 150);

**Customer** mary = **new** **Customer**("Mary T.", 1);

List<Customer> customers = Arrays.asList(john, sarah, charles, mary);

### **2.1. Filtering Collections**

A common use case of the filter() method is [processing collections](https://www.baeldung.com/java-collection-filtering).

Let's make a list of customers with more than 100 points. To do that, we can use a lambda expression:

List<Customer> customersWithMoreThan100Points = customers

.stream()

.filter(c -> c.getPoints() > 100)

.collect(Collectors.toList());

We can also use a [method reference](https://www.baeldung.com/java-8-double-colon-operator), which is shorthand for a lambda expression:

List<Customer> customersWithMoreThan100Points = customers

.stream()

.filter(Customer::hasOverHundredPoints)

.collect(Collectors.toList());

In this case, we added the hasOverHundredPoints method to our Customer class:

**public** **boolean** **hasOverHundredPoints**() {

**return** this.points > 100;

}

In both cases, we get the same result:

assertThat(customersWithMoreThan100Points).hasSize(2);

assertThat(customersWithMoreThan100Points).contains(sarah, charles);

### **2.2. Filtering Collections with Multiple Criteria**

Furthermore, we can use multiple conditions with filter(). For example, we can filter by points and name:

List<Customer> charlesWithMoreThan100Points = customers

.stream()

.filter(c -> c.getPoints() > 100 && c.getName().startsWith("Charles"))

.collect(Collectors.toList());

assertThat(charlesWithMoreThan100Points).hasSize(1);

assertThat(charlesWithMoreThan100Points).contains(charles);

## **3. Handling Exceptions**

Until now, we've been using the filter with predicates that don't throw an exception. Indeed, the **functional interfaces in Java don't declare any checked or unchecked exceptions**.

Next we're going to show some different ways to handle [exceptions in lambda expressions](https://www.baeldung.com/java-lambda-exceptions).

### **3.1. Using a Custom Wrapper**

First, we'll start by adding a profilePhotoUrl to our Customer:

**private** String profilePhotoUrl;

In addition, let's add a simple hasValidProfilePhoto() method to check the availability of the profile:

**public** **boolean** **hasValidProfilePhoto**() **throws** IOException {

**URL** url = **new** **URL**(this.profilePhotoUrl);

**HttpsURLConnection** connection = (HttpsURLConnection) url.openConnection();

**return** connection.getResponseCode() == HttpURLConnection.HTTP\_OK;

}

We can see that the hasValidProfilePhoto() method throws an IOException. Now if we try to filter the customers with this method:

List<Customer> customersWithValidProfilePhoto = customers

.stream()

.filter(Customer::hasValidProfilePhoto)

.collect(Collectors.toList());

We'll see the following error:

Incompatible thrown types java.io.IOException in functional expression

To handle it, one of the alternatives we can use is wrapping it with a try-catch block:

List<Customer> customersWithValidProfilePhoto = customers

.stream()

.filter(c -> {

**try** {

**return** c.hasValidProfilePhoto();

} **catch** (IOException e) {

//handle exception

}

**return** false;

})

.collect(Collectors.toList());

If we need to throw an exception from our predicate, we can wrap it in an unchecked exception like RuntimeException.

### **3.2. Using ThrowingFunction**

Alternatively, we can use the ThrowingFunction library.

ThrowingFunction is an open source library that allows us to handle checked exceptions in Java functional interfaces.

Let's start by adding the [throwing-function dependency](https://search.maven.org/search?q=g:pl.touk%20AND%20a:throwing-function%26core%3Dgav) to our pom:

<**dependency**>

<**groupId**>pl.touk</**groupId**>

<**artifactId**>throwing-function</**artifactId**>

<**version**>1.3</**version**>

</**dependency**>

To handle exceptions in predicates, this library offers us the ThrowingPredicate class, which has the unchecked() method to wrap checked exceptions.

Let's see it in action:

**List** customersWithValidProfilePhoto = customers

.stream()

.filter(ThrowingPredicate.unchecked(Customer::hasValidProfilePhoto))

.collect(Collectors.toList());

JAVA TIME in Java 8:\*\*\*\*\*\*\*\*\*\*\*

Java 8 introduced new APIs for *Date* and *Time* to address the shortcomings of the older *java.util.Date* and *java.util.Calendar*.

let's start with the issues in the existing *Date* and *Calendar* APIs and discuss how the new Java 8 *Date* and *Time* APIs address them.

We will also look at some of the core classes of the new Java 8 project that are part of the *java.time* package, such as *LocalDate*, *LocalTime*,*LocalDateTime*,*ZonedDateTime*,*Period*,*Duration*and their supported APIs.

**Issues With the Existing *Date*/*Time*APIs**

* **Thread safety** – The *Date* and *Calendar* classes are not thread safe, leaving developers to deal with the headache of hard-to-debug concurrency issues and to write additional code to handle thread safety. On the contrary, the new *Date* and *Time* APIs introduced in Java 8 are immutable and thread safe, thus taking that concurrency headache away from developers.
* **API design and ease of understanding** – The *Date* and *Calendar* APIs are poorly designed with inadequate methods to perform day-to-day operations. The new *Date*/*Time*API is ISO-centric and follows consistent domain models for date, time, duration and periods. There are a wide variety of utility methods that support the most common operations.
* ***ZonedDate* and *Time*** – Developers had to write additional logic to handle time-zone logic with the old APIs, whereas with the new APIs, handling of time zone can be done with *Local* and *ZonedDate*/*Time* APIs.

## . Using LocalDate, LocalTime and LocalDateTime

The most commonly used classes are LocalDate, LocalTime and LocalDateTime. As their names indicate, they represent the local date/time from the context of the observer.

We mainly use these classes when time zones are not required to be explicitly specified in the context. As part of this section, we will cover the most commonly used APIs.

### **Working With**LocalDate

The LocalDate represents **a date in ISO format (yyyy-MM-dd) without time.**We can use it to store dates like birthdays and paydays.

An instance of current date can be created from the system clock:

**LocalDate** localDate = LocalDate.now();

And we can get the LocalDate representing a specific day, month and year by using the of method or the parse method.

For example, these code snippets represent the LocalDate for February 20, 2015:

LocalDate.of(2015, 02, 20);

LocalDate.parse("2015-02-20");

The LocalDate provides various utility methods to obtain a variety of information. Let's have a quick peek at some of these API methods.

The following code snippet gets the current local date and adds one day:

**LocalDate** tomorrow = LocalDate.now().plusDays(1);

This example obtains the current date and add one day. Note how it accepts an *enum* as the time unit:

**LocalDate** previousMonthSameDay = LocalDate.now().minus(1, ChronoUnit.MONTHS);

In the following two code examples, we parse the date “2016-06-12” and get the day of the week and the day of the month respectively. Note the return values — the first is an object representing the *DayOfWeek*, while the second is an *int* representing the ordinal value of the month:

**DayOfWeek** sunday = LocalDate.parse("2016-06-12").getDayOfWeek();

**int** twelve = LocalDate.parse("2016-06-12").getDayOfMonth();

We can test if a date occurs in a leap year, for example the current date:

**boolean** leapYear = LocalDate.now().isLeapYear();

Also, the relationship of a date to another can be determined to occur before or after another date:

**boolean** notBefore = LocalDate.parse("2016-06-12")

.isBefore(LocalDate.parse("2016-06-11"));

**boolean** isAfter = LocalDate.parse("2016-06-12")

.isAfter(LocalDate.parse("2016-06-11"));

Finally, date boundaries can be obtained from a given date.

### **Working With**LocalTime

The LocalTime represents **time without a date.**

Similar to LocalDate, we can create an instance of LocalTime from the system clock or by using parse and of methods.

We'll now take a quick look at some of the commonly used APIs.

An instance of current LocalTime can be created from the system clock:

**LocalTime** now = LocalTime.now();

### **Working With**LocalDateTime

LocalDateTime is used to represent **a combination of date and time.** This is the most commonly used class when we need a combination of date and time.

The class offers a variety of APIs. Here, we'll look at some of the most commonly used ones.

An instance of LocalDateTime can be obtained from the system clock similar to LocalDate and LocalTime:

LocalDateTime.now();

The below code samples explain how to create an instance using the factory “of” and “parse” methods. The result would be a LocalDateTime instance representing February 20, 2015, 6:30 a.m.:

LocalDateTime.of(2015, Month.FEBRUARY, 20, 06, 30);

LocalDateTime.parse("2015-02-20T06:30:00");

There are utility APIs to support addition and subtraction of specific units of time like days, months, years and minutes.

The below code demonstrates the “plus” and “minus” methods. These APIs behave exactly like their counterparts in LocalDate and LocalTime:

## Using ZonedDateTime API

Java 8 provides *ZonedDateTime*when we need to deal with time-zone-specific date and time. The ZoneId is an identifier used to represent different zones. There are about 40 different time zones, and the ZoneId represents them as follows.

Here, we create a Zone for Paris:

**ZoneId** zoneId = ZoneId.of("Europe/Paris");

And we can get a set of all zone ids:

Set<String> allZoneIds = ZoneId.getAvailableZoneIds();

The LocalDateTime can be converted to a specific zone:

**ZonedDateTime** zonedDateTime = ZonedDateTime.of(localDateTime, zoneId);

## Using Period and Duration

The Period class represents a quantity of time in terms of years, months and days, and the Duration class represents a quantity of time in terms of seconds and nanoseconds.

### **5.1. Working With**Period

The Period class is widely used to modify values of given a date or to obtain the difference between two dates:

**LocalDate** initialDate = LocalDate.parse("2007-05-10");

We can manipulate the Date by using Period:

**LocalDate** finalDate = initialDate.plus(Period.ofDays(5));

The Period class has various getter methods such as getYears, getMonths and getDays to get values from a Period object.

### **Working With**Duration

Similar to Period, the Duration class is used to deal with Time.

Let's create a LocalTime of 6:30 a.m. and then add a duration of 30 seconds to make a LocalTime of 6:30:30 a.m.:

**LocalTime** initialTime = LocalTime.of(6, 30, 0);

**LocalTime** finalTime = initialTime.plus(Duration.ofSeconds(30));

## Compatibility With Date and Calendar

Java 8 has added the toInstant() method, which helps to convert existing Date and Calendar instance to new Date and Time API:

LocalDateTime.ofInstant(date.toInstant(), ZoneId.systemDefault());

LocalDateTime.ofInstant(calendar.toInstant(), ZoneId.systemDefault());

## Date and Time Formatting

Java 8 provides APIs for the easy formatting of Date and Time:

**LocalDateTime** localDateTime = LocalDateTime.of(2015, Month.JANUARY, 25, 6, 30);

This code passes an ISO date format to format the local date, with a result of 2015-01-25:

**String** localDateString = localDateTime.format(DateTimeFormatter.ISO\_DATE);

## Backport and Alternate Options

### **8.1. Using the ThreeTen Project**

For organizations that are on the path of moving to Java 8 from Java 7 or Java 6 and that want to use date and time API, the [ThreeTen](http://www.threeten.org/) project provides the backport capability.

Developers can use classes available in this project to achieve the same functionality as that of new Java 8 Date and Time APIs. And once they move to Java 8, the packages can be switched.

The artifact for the ThreeTen project can be found in the [Maven Central Repository](https://mvnrepository.com/artifact/org.threeten/threetenbp):

<**dependency**>

<**groupId**>org.threeten</**groupId**>

<**artifactId**>threetenbp</**artifactId**>

<**version**>1.3.1</**version**>

</**dependency**>

### **8.2. Joda-Time Library**

Another alternative for Java 8 Date and Time library is [Joda-Time](http://www.joda.org/joda-time/) library. In fact, the Java 8 Date/Time API has been led jointly by the author of Joda-Time library (Stephen Colebourne) and Oracle. This library provides pretty much all capabilities that are supported in the Java 8 Date/Time project.

The artifact can be found in [Maven Central](https://mvnrepository.com/artifact/joda-time/joda-time) by including the below pom dependency in our project:

<**dependency**>

<**groupId**>joda-time</**groupId**>

<**artifactId**>joda-time</**artifactId**>

<**version**>2.9.4</**version**>

</**dependency**>