Lambda :In these cases, you're usually trying to pass functionality as an argument to another method, such as what action should be taken when someone clicks a button.

**java.util.Stream**

# Stream In Java

Introduced in Java 8, the 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-  
**Intermediatory Operations:**

1. **map:** The map method is used to map the items in the collection to other objects according to the Predicate passed as argument.  
   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=ames.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)

To perform functional style operations on elements

**Fork and join** in stream i.e break task into subtasks (fork) and combine result (join)

Q) list of students. Retrieve students whose name starts with A

List names=students.stream().map(s-> s.getName()).filter(name -> name.startsWith("A")).collect(Collectors.toList());

Distinct()-> override equals

**Collections vs streams:**

Colelctions are in-memory data structures which hold elements within it. Each element in the collection is computed before it actually becomes part of the collection. Whereas streams are fixed data structures which computes elements on demand basis.

Java 8 streams can be seen as lazily constructed collection where values are computed when user demands for it. Streams do not store elements, it simply conveys elements from source through a pipeline of computational operations.

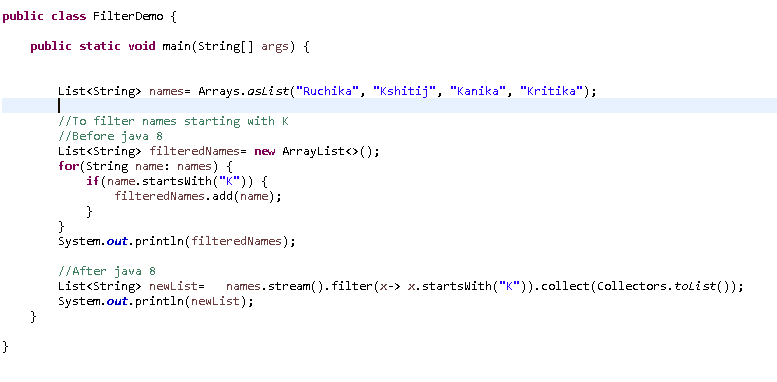
Streams are functional in nature. Operations performed on stream does not modify it’s source. Eg. Filtering produces a new stream rather than removing elements from source.

Methods will start returning streams. The order is maintained as source. Supports aggregate operations (filter, map, reduce, find, match, sort.. which can be executed in parallel or series)

Streams also support pipeline and internal iterations. Streams have internal iterating methods like filter, map, forEach etc. where code is completely unaware of iteration logic in background.

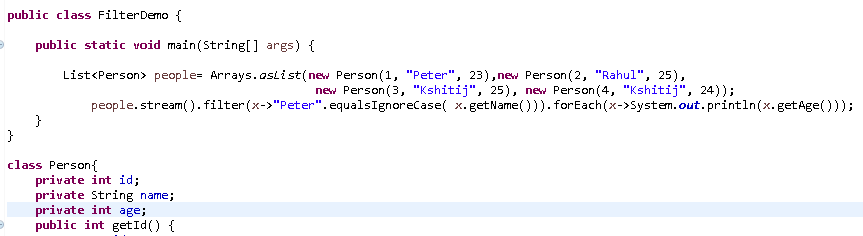
We can use stream to filter, collect, print and convert from one data structure to other.

Stream interface

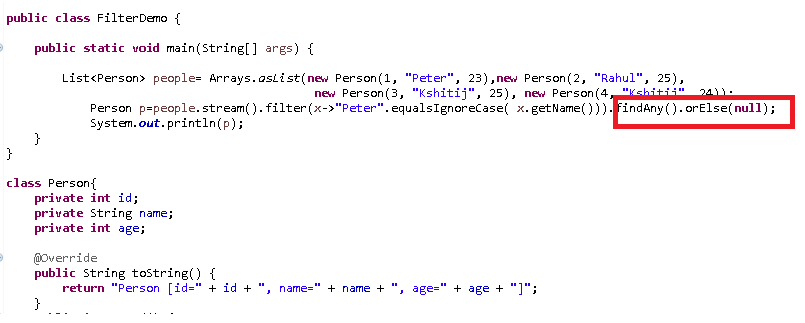


Q) Class Person {id, name, age}

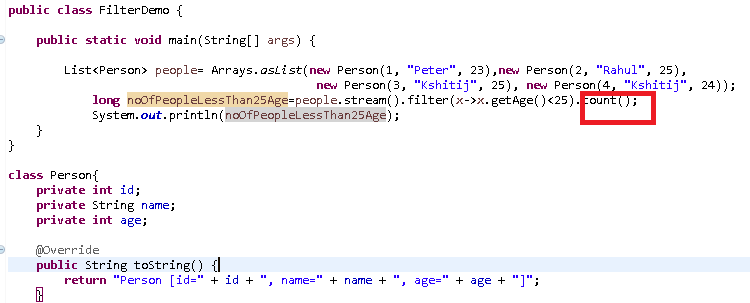
Filter list of persons to find age of Peter



Q) Find person with name Peter from list of persons

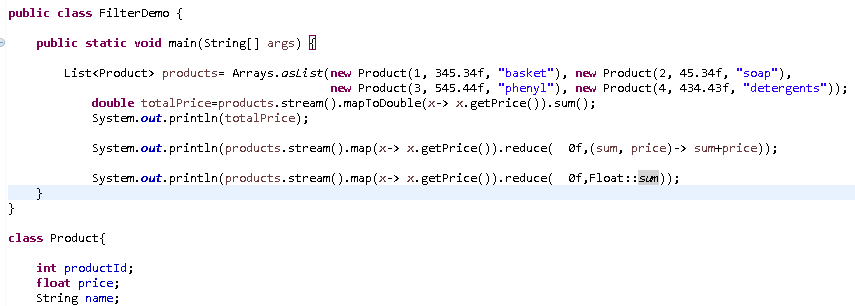


Q) Find number of people with age <25

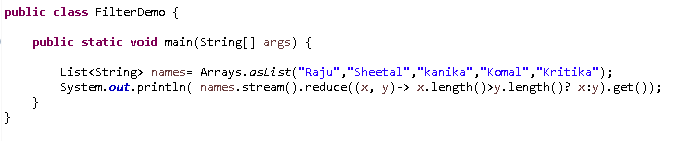


count(): returns number of elements in the stream

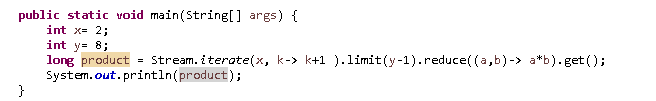
Q) Find sum of all product prices



Q) Write a program to find the largest string



Q) Print product of all numbers in a given range



**Stream.reduce() with Accumulator**

Here we will pass BinaryOperator as accumulator. In case of numeric BinaryOperator, the start value will be 0. In case of string, the start value will be a blank string.   
The method will return Optional instance

### Stream.reduce() with Identity and Accumulator

Here will use an identity and accumulator. We will pass the identity as start value.

**Stream.reduce() with Identity, Accumulator and Combiner**

Here we will pass three arguments identity, accumulator and combiner in reduce() method. This method with these three arguments is used in parallel processing. Combiner works with parallel stream only, otherwise there is nothing to combine.   
  
reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)  
  
Find the example.   
**ReduceDemo3.java**

package com.concretepage;

import java.util.Arrays;

import java.util.List;

public class ReduceDemo3 {

public static void main(String[] args) {

List<Integer> list2 = Arrays.asList(2, 3, 4);

//Here result will be 2\*2 + 2\*3 + 2\*4 that is 18.

int res = list2.parallelStream().reduce(2, (s1, s2) -> s1 \* s2, (p, q) -> p + q);

System.out.println(res);

List<String> list1 = Arrays.asList("Mohan", "Sohan", "Ramesh");

String result = list1.parallelStream().reduce("-", (s1, s2) -> s1 + s2, (p, q) -> p + q);

System.out.println(result);

}

}

**Output**

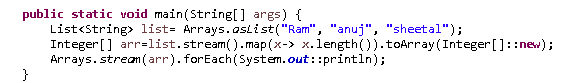
18

-Mohan-Sohan-Ramesh

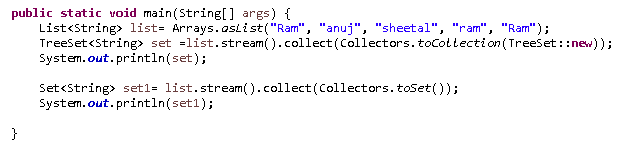
|  |  |  |  |
| --- | --- | --- | --- |
| **Stream.reduce() with Identity, Accumulator and Combiner**  It performs a reduction on the elements of the given stream, using given identity value, an accumulation function and combining functions. The identity value must be an identity for the combiner function. Combiner is a function which aggregates results of the accumulator. Combiner is called only in a parallel mode to reduce results of accumulators from different threads.   |  |  |  | | --- | --- | --- | | [?](http://www.java2novice.com/java-8/streams/reduce-method-example/)   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package com.java2novice.streams;    import java.util.stream.Stream;    public class StreamReduceAccEx {        public static void main(String[] args) {            Integer arrSum = Stream.of(10,20,22,12,14).parallel().reduce(1000, (x,y)->x+y, (p,q)->{              System.out.println("combiner called");              return p+q;          });          System.out.println(arrSum);      }  } | | |
| |  | | --- | | Output: | | combiner called  combiner called  combiner called  combiner called  5078 | |

Collect method

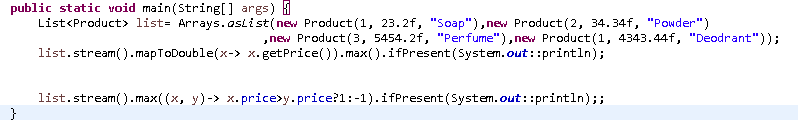
Q) Find length of elements of the list and return the result in an array



Q) Save the elements in the list to Treeset and set respectively



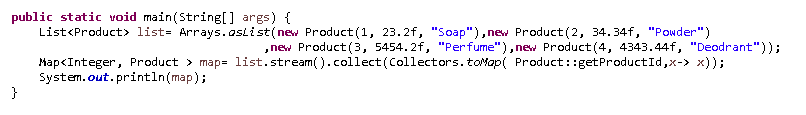
Q) Find product with max price



Output:

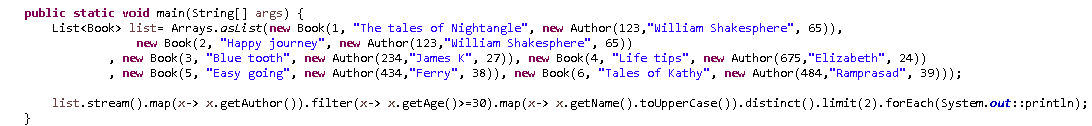


Q) Convert product list to map

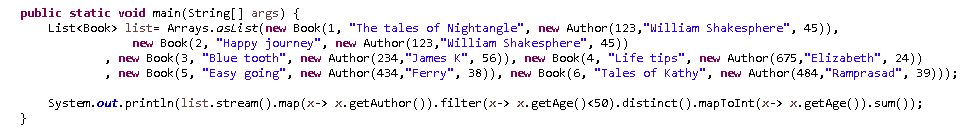


If duplicate key, throws IllegalStateException

Q) Get unique names in uppercase of first 2 book authors that are 30 years old or older



Q) Get sum of ages of all authors younger than 50 years



Q) Concatenate string elements of the list



**Short circuit operations of stream:** processes elements till it finds first match

* 1. anyMatch(Predicate condition): checks if any element matches condition. If yes, returns true.



* 1. findFirst:



* 1. limit(int number)
  2. findAny():Returns an Optional instance which wraps any and only one element of the stream. it is free to select any element in the stream. This is to allow for maximal performance in parallel operations; the cost is that multiple invocations on the same source may not return the same result. (If a stable result is desired, use findFirst() instead. For a sequential stream there won't be any difference between 'findFirst' and 'findAny'. But for a parallel stream findAny will return 'any' element rather than waiting for the 'first' element.
  3. allMatch(Predicate ): Tests whether all elements match the provided predicate. It may return early with false result when any element doesn't match first.
  4. boolean noneMatch(Predicate<? super T> predicate):Tests whether no elements of this stream match the provided predicate. It may return early with false result when any element matches the provided predicate first.

**Parallel Streams**

Collection.stream()

Collection.parallelStream()

list.stream()

list.parallelStream()

map.entrySet().stream()

map.entrySet().parallelStream()

IntStream.range(int, int)

LongStream

DoubleStream

****

**Lazy Processing**

Stream.of("d2", "a2", "b1", "b3", "c")

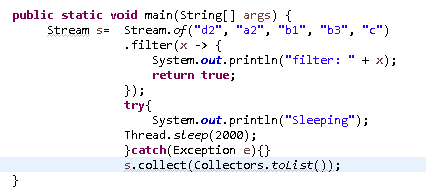
.filter(s -> {

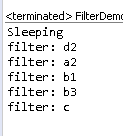
System.out.println("filter: " + s);

return true;

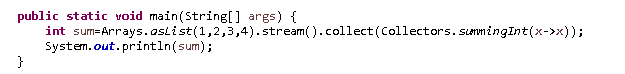
});

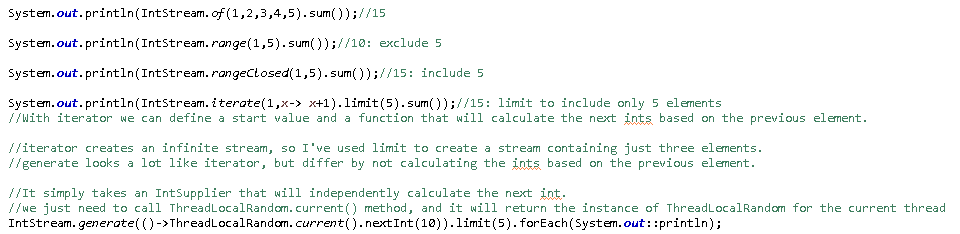
When executing this code snippet, nothing is printed to the console. That is because intermediate operations will only be executed when a terminal operation is present.



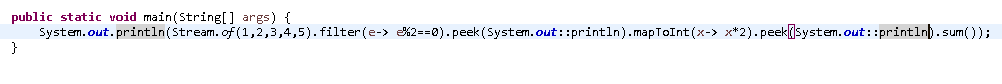


**Collect(Collectors.summingInt()): will do the sum of stream elements**

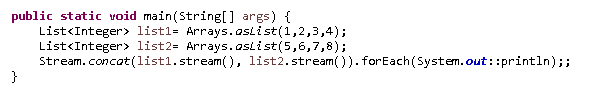
****

****

**Peek(): To support debugging, to see elements in the stream as they flow past the pipeline**

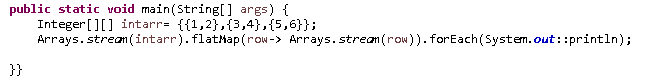
****

**Stream concatenation:**

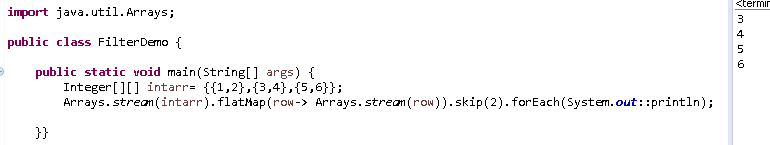


**Flatmap():**a flatMap() is used to convert a Stream of Stream into a list of values

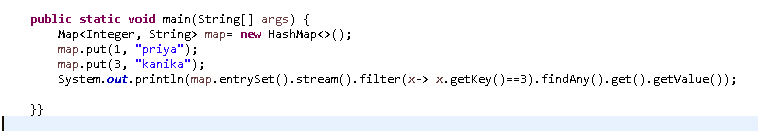
It returns a stream of object after applying mapping function on each element and then flattens the result



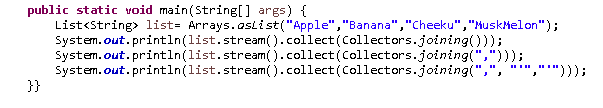
**Skip():** It returns the stream skipping the given number of elements

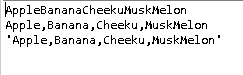


Q) Filter a map and return the value



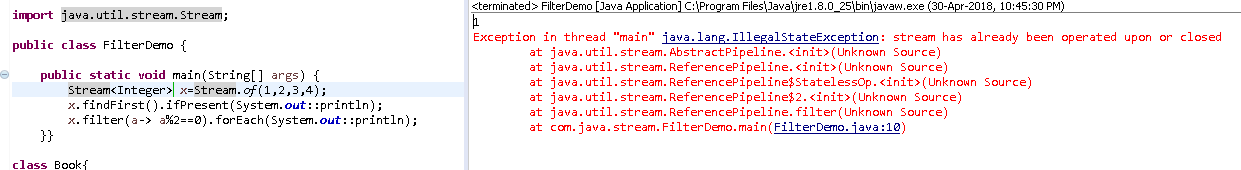
**Collectors.joining():** Can also pass delimiter, prefix, suffix in brackets





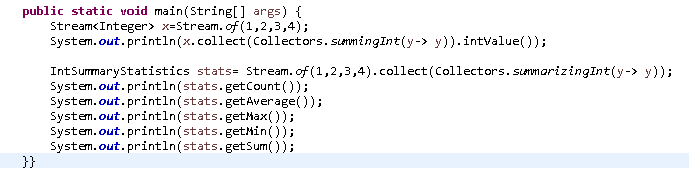
Q) Convert stream of set to stream of string

**Stream reuse:** Attempt to reuse the same stream after calling terminal operation will trigger IllegalStateException.



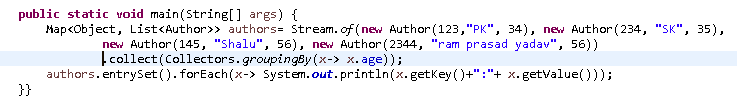
Coz streams are not meant to store values.

**AveragingInt(), SumingInt():** return sum of all elements of stream, SummarizingInt(): collect statistical info about stream elements

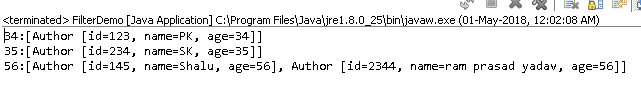




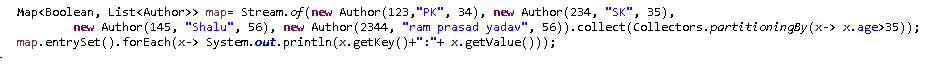
**groupingBy:** grouping of stream elements based on some function



o/p:



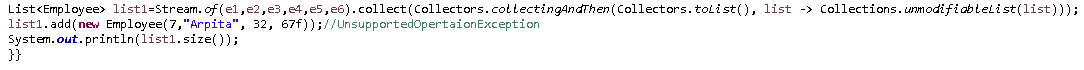
**partitioningBy** : divide elements of stream into groups according to some predicate (true or false). Map key is Boolean, value is list of elements



o/p: 

**CollectingAndThen method**

Pushing collector to perform additional transformation

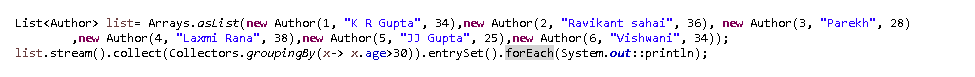


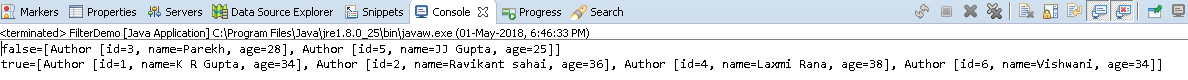
**ParallelStream:**

stream.sorted().parallel()

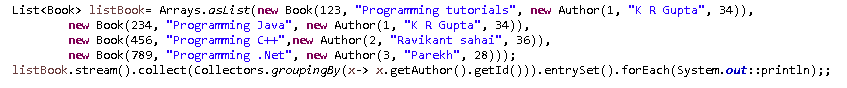
list.parallelStream()

**Grouping:**





Grouping based on author id:





**Closures in Lambda expressions:**

Cannot change value of outer class variable in inner anonymous class. Variable is implicitly final

**Optional class:**

It helps in preventing NullPointerException. Container of object of type T. When value is null, allows us to perform some pre-defined actions instead of throwing NPE.

Instance of it can be created with the help of the static method:

Optional<String> optional= Optional.empty();

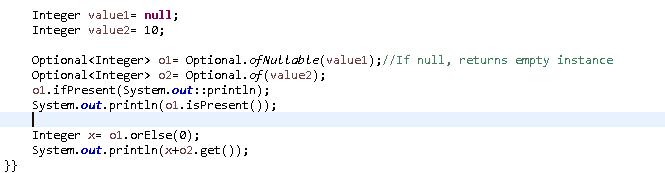
String str=”value”;

Optional o= Optional.of(str);

Returns an optional which contains a not-null value:

Optional<String> o= Optional.ofNullable(getString());

Returns specified value or empty optional if parameter is null.



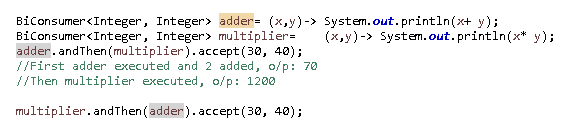
o/p

false

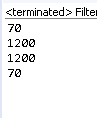
10

**Biconsumer:**

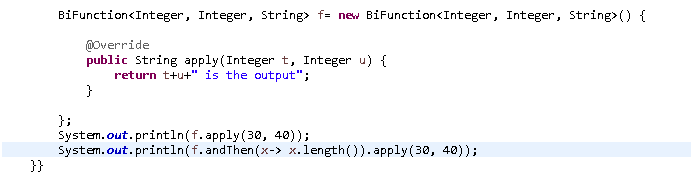
Takes in two object, returns nothing. Map ‘forEach’ method takes in biconsumer object.



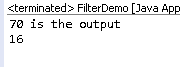
o/p:



**BiFunction:**  Takes in two arguments and returns a value. Has default andThen method.

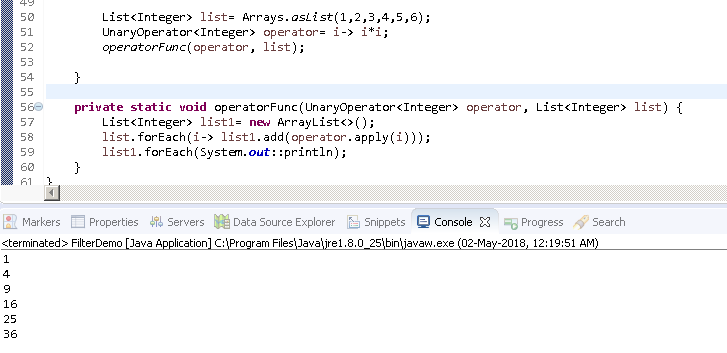


**o/p:**

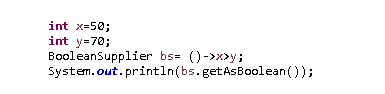
****

**BiPredicate**

**UnaryOperator:** Extends function interface. No abstract method. UnaryOperator extends Function and BinaryOperator extends BiFunction and accordingly they accept argument. UnaryOperator accepts one operand and returns a value of the same type as operand. BinaryOperator accepts two operand of the same type and returns the result of the same type as operand.



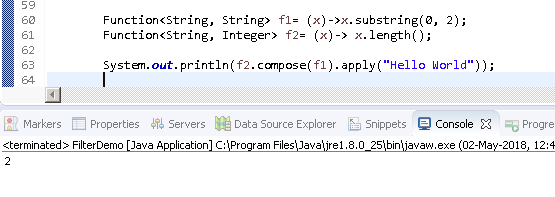
**BooleanSupplier**: Supplies Boolean value. Can return same value everytime. Has one method getAsBoolean() which returns Boolean value.



o/p: false

**Function default methods: compose, andThen and Identity:**

**Compose:** First applies the function to it’s input and then applies the function to the result. If evaluating either throws an exception, it is relayed to the caller of the composed function.



The difference between compose and andThen is the order they execute the functions. While the compose function executes the caller last and the parameter first, the andThen executes the caller first and the parameter last

<https://dzone.com/articles/higher-order-functions>

Function.identity(): returns whatever passed to argument.

Map.compute(): If this function returns null, entry for key is removed from map, if key not present, entry added in map.



**Map.merge():**

Merge the specified value to the existing Value using the Specified function for the Specified Key.  
If the Key is not present , Add an element with the new Key and Value.

It associates given value with given key using a given mapping function. Non-null value to be merged with existing value associated to the key. If no existing value or is a null value, to be added to the key. The new value associated with given key is returned or null if no value is associated to the key.

UnsupportedOperationException if operation is not performed by this map

ClassCastException , if the class of key/ value prevents it from being stored in map

NullPointerException: If specified key is null and map does not support null keys or value or remapping function is null.

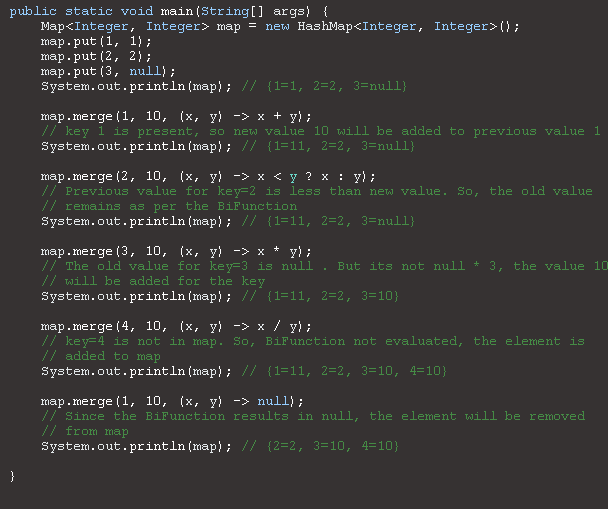
If function returns null, mapping for key is removed from map

Example  
  
strMap.merge("Key1","Value56",(v1,v2)->v1.substring(3).concat(v2));  
  
Sample Code 1  
  
Map<String,String> intMap = new HashMap<String,String>();  
strMap.put("Key1","Value1");  
strMap.put("Key2", "Value2");     
String str = strMap.merge("Key1","Value56",(v1,v2)->v1.substring(3).concat(v2));  
System.out.println(str); // prints ue1Value56  
System.out.println(strMap); // prints {Key2=Value2, Key1=ue1Value56}  
  
Sample Code 2   
  
Map<String,String> strMap = new HashMap<String,String>();  
strMap.put("Key1","Value1");  
strMap.put("Key2", "Value2");     
String str = strMap.merge("Key4","Value56",(v1,v2)->v1.substring(3).concat(v2));  
System.out.println(str); // prints Value56  
System.out.println(strMap); // prints {Key2=Value2, Key1=Value1, Key4=Value56}

Summary:

If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.

Otherwise, replaces the associated value with the results of the given remapping function, or removes if the result is null. This method may be of use when combining multiple mapped values for a key.



**Permgen to metaspace:**

Permgen space removed in java8. Metaspace is replacement of permgen. Class metadata allocated in native memory. The amount of memory that can be used for class metadata is by default unlimited. As permgen has been removed completely and has been replaced by new space i.e. Metaspace, so permsize and permgensize are ignored completely and will not throw OutOfMemoryError:PermGen error.

PermGen is an abbreviation for Permanent Generation and it’s a special heap space which is separate from the main Java heap where JVM keeps track of metadata of the classes which have been loaded. In Java 8, PermGen has been renamed to Metaspace - with some subtle differences. From our perspective, it is important to note that Metaspace has an unlimited default maximum size. On the contrary, PermGen from Java 7 and earlier has a default maximum size of 64 MB on 32-bit JVM and 82 MB on the 64-bit version. Of course, these are not the same as the initial sizes. Java 7 and earlier starts with something around 12-21 MB of the initial PermGen space.

|  |  |  |
| --- | --- | --- |
| JVM | Default maximum PermGen size (MB) | Default maximum Metaspace size |
| 32-bit client JVM | 64 | unlimited |
| 32-bit server JVM | 64 | unlimited |
| 64-bit JVM | 82 | unlimited |

It is worthwhile to mention that prior to Java 7, interned Strings used to be kept on the PermGen. That caused some serious problems with the infamous:

java.lang.OutOfMemoryError: PermGen space

Whenever there is a need to resize PermGen/Metaspace, JVM will do it as it does with the standard heap. Resizing those spaces requires a full GC, which is always an expensive operation. It can usually be observed during a startup when a lot of classes are being loaded. Especially if the application has dependencies on many external libraries. If there are a lot of full GCs during the startup, it’s usually because of that. If that case, increasing the initial size can boost the startup performance.

To increase PermGen, we have the following commands:

-XX:PermSize=N -  sets the initial (and minimum size) of the Permanent Generation space.

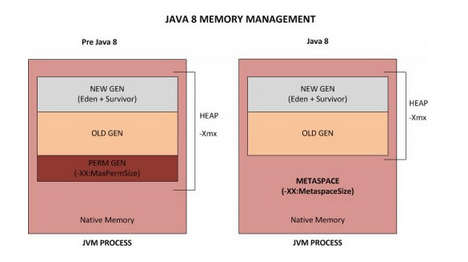
-XX:MaxPermSize=N  -  sets the maximum size of the Permanent Generation space.

In Java 8 and onwards, we can set the initial and maximum size of Metaspace using the following commands:

-XX:MetaspaceSize=N  - sets the initial (and minimum size) of the Metaspace.

-XX:MaxMetaspaceSize=N  - sets the maximum size of the Metaspace.

**Metaspace is NOT part of Heap.** Rather Metaspace is part of **Native Memory (process memory)** which is only limited by the Host Operating System.



**So, what is the significance of this change?**

While you will NOT run out of PermGen space anymore (since there is NO PermGen), you may consume excessive Native memory making the total process size large. The issue is, if your application loads lots of classes (and/or interned strings), **you may actually bring down the Entire Server** (not just your application). Why ? Because the native memory is only limited by the Operating System. This means you can literally take up all the memory on the Server. Not good.

It is critical that you add the new option **-XX:MaxMetaspaceSize**  which sets the Maximum Metaspace size for your application.

Note that it is no longer sufficient to just monitor the Heap Size. You must also monitor the Metaspace which you can do by just **keeping an eye on the ‘process size’** using your Operating System utilities (Example: ‘top’ in Unix/Linux, ‘Task Manager’ in Windows).

**Bonus Tip:**

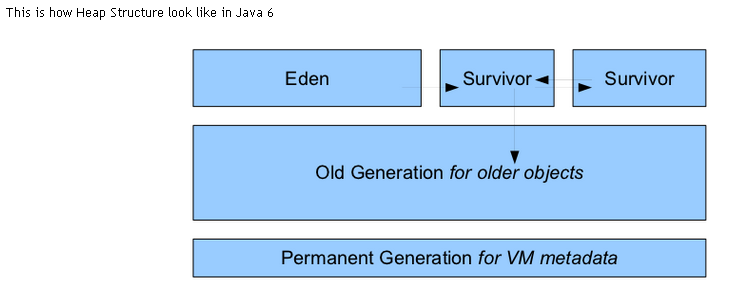
You can use the **jmap** command to print out Memory statistics of your current pre Java 8 application.

**jmap -permstat <PID>**

**MetaspaceSize :** This flag comes into the picture when you want to change the default size of your native memory that jvm can use. Use this flag only when you are sure that your application is going to need more memory than what is provided by default otherwise you will end up in suppressing some of the Garbage Collectors that were supposed to run in the default configuration.

**MaxMetaspaceSize :** This flag provides you flexibility to set the maximum size that you want to allocate for metaspace. You might want to do this to control the space used by your applications (multiple) running on a server or in a case of memory leak so that it won’t use all of available space in native memory. Now in the case when your native memory is fully and your application still needs more space in that case you will see a new error message : **java.lang.OutOfMemoryError: Metadata space**

Only advantage now you will have more space in the heap, but on the other side we don’t have any tool with which we can analyze this meta-data information as compared to the tools available e.g.jconsole ,HeapAnalyzer etc for analyzing heap.



**Why was PermGen Eliminated?**

* Fixed size at startup – difficult to tune.
  + -XX:MaxPermSize=?
* Internal Hotspot types were Java objects : Could move with full GC, opaque, not strongly typed and hard to debug, needed meta-metadata.
* Simplify full collections : Special iterators for metadata for each collector
* Want to deallocate class data concurrently and not during GC pause
* Enable future improvements that were limited by PermGen.