Old way to find min()

```
int [] numbers = {4, 1, 13, 90, 16, 2, 0};

int [] min = numbers[0];
for (int i = 1; I < numbers.length; i++) {
     if (min < numbers[i] {
         min = numbers[i];
     }
}
System.out.println("Minimum is  " + min);
```

===> with Java 8 Streams need just one line

```
IntStream.of (numbers).min();
\
 \
  )))))))))>   int min = IntStream.of (numbers)
                         .min()
                         .getAsInt();
                                  Will throw exception if min cannot be found (ex, if array is empty)

  )))))))))>   int min = IntStream.of (numbers)
                         .min()
                         .ifPresent(System.out::println);
```

more funcs min(), max(), average(), count(), sum()

Old way to find 3 distinct smallest numbers

```
int [] numbers = {4, 1, 13, 90, 16, 2, 0};

// clone to avoid mutating original array
int [] copy = Arrays.copyOf(numbers, numbers.length);

// sort
Arrays.sort(copy);

// pick first 3
for (int i = 0; I < 3; i++) {
     // need few more lines for distinct numbers
     // ….
     System.out.println(copy[i]);
}
```

```
\
 \
  )))))))))>   IntStream.of (numbers)                  < ---------- 1, Create
                .distinct()              ]
                .sorted                  }      < ---------- 2, Process
                .limit(3)                ]
                .forEach(System.out::println);     < -------- 3, Consume
```

```java
IntStream.of(numbers).distinct();                  // distinct
IntStream.of(numbers).sorted();                    // sort
IntStream.of(numbers).limit(3);                    // get first 3
IntStream.of(numbers).skip(3);                     // skip first 3
IntStream.of(numbers).filter(num -> num % 2 == 0); // only even
IntStream.of(numbers).map(num -> num * 2);         // double each num
IntStream.of(numbers).boxed();                     // convert each num to Integer
IntStream.of(numbers).average(); // average
IntStream.of(numbers).min();     // min
IntStream.of(numbers).max();     // max
IntStream.of(numbers).sum();     // sum
IntStream.of(numbers).count();   // count

IntStream.range(1, 100).forEach(System.out::println);         // print 1 to 99
IntStream.range(1, 100).toArray();                            // collect into array
IntStream.range(1, 100).boxed().collect(Collectors.toList()); // collect into list

IntStream.of(numbers).anyMatch(num -> num % 2 == 1);  // is any num odd
IntStream.of(numbers).allMatch(num -> num % 2 == 1);  // are all num odd
```

Old way to get names of 3 highest earning employees

```java
List<Employee> employees = getAllEmployees();

// New list
List<Employee> copy = new ArrayList<>(employees);

// Sort descending
copy.sort((o1, o2) -> o2.getSalary() - o1.getSalary());

// Get first 3
for (int i = 0; i < 3; i++) {
    Employee employee = copy.get(i);
    System.out.println(employee.getName());
}
```

Avoid mutating original array

\
))))))))>

```java
List<Employee> employees = getAllEmployees();
employees.stream()
        .sorted(Comparator.comparingInt(Employee::getSalary).reversed())
        .limit(3)
        .map(Employee::getName)
        .forEach(System.out::println);
```

Can also do .map(emp -> emp.getName())

Multiple comparators are added in Java 8!

Enhanced above with filter()

```
List<Employee> employees = getAllEmployees();
List<String> names
        = employees.stream()
            .sorted(Comparator.comparingInt(Employee::getSalary).reversed())
            .filter(employee -> isActive(employee))
            .limit(3)
            .map(Employee::getName)
            .collect(Collectors.toList());
```

Finally collect result into a list

Filter/Keep ones that are active

Collectors are awesome!

```
// to list
List<String> listOfEmps
        = employees.stream()
            .limit(3)
            .map(Employee::getName)
            .collect(Collectors.toList());

// to set
Set<String> setOfEmps
        = employees.stream()
            .limit(3)
            .map(Employee::getName)
            .collect(Collectors.toSet());
```

```
// to map
Map<String, Employee> empMap
        = employees.stream()
            .limit(3)
            .collect(Collectors.toMap(e -> e.name, e -> e));
```

```
// john, amy, marcy
String names
        = employees.stream()
            .limit(3)
            .map(Employee::getName)
            .collect(Collectors.joining( delimiter: ", "));

// group by dept
Map<String, List<Employee>> empByDept
        = employees.stream()
            .collect(Collectors.groupingBy(e -> e.dept));

// count employees in each dept
Map<String, Long> deptCounts
        = employees.stream()
            .collect(Collectors.groupingBy(Employee::getDept, Collectors.counting()));
```

```
// parallel streams
Map<String, List<Employee>> empByDep
        = employees.stream()
            .parallel() // very easy and deceptive
            .collect(Collectors.groupingBy(e -> e.dept));
```

1. Introduction

The Stream API was one of the key features added in Java 8.

Briefly, the API allows us to process collections and other sequences of elements – conveniently and more efficiently – by providing a declarative API.

2. Primitive Streams

Streams primarily work with collections of objects and not primitive types.

Fortunately, to provide a way to work with the three most used primitive types – *int*, *long* and *double* – the standard library includes three primitive-specialized implementations: *IntStream*, *LongStream*, and *DoubleStream*.

Primitive streams are limited mainly because of boxing overhead and because creating specialized streams for other primitives isn't' that useful in many cases.

3. Arithmetic Operations

Let's start with a few interesting methods for heavily used arithmetic operations such as *min*, *max*, *sum*, and *average:*

```
1   int[] integers = new int[] {20, 98, 12, 7, 35};
2   int min = Arrays.stream(integers)
3     .min()
4     .getAsInt(); // returns 7
```

Let's now step through the code snippet above to understand what's going on.

We created our *IntStream* by using *java.util.Arrays.stream(int[])* and then used the *min()* method to get the lowest integer as *java.util.OptionalInt* and finally called *getAsInt()* to get the *int* value.

Another way to create an *IntStream* is using *IntStream.of(int…)*. The *max()* method will return the greatest integer:

```
1   int max = IntStream.of(20, 98, 12, 7, 35)
2     .max()
3     .getAsInt(); // returns 98
```

Next – to get the sum of integers we just call the *sum()* method and we don't need to use *getAsInt()* since it already returns the result as an *int* value:

```
1   int sum = IntStream.of(20, 98, 12, 7, 35).sum(); // returns 172
```

We invoke the *average()* method to get the average of integer values and as we can see, we should use *getAsDouble()* as it returns a value of type *double*.

```
double avg = IntStream.of(20, 98, 12, 7, 35)
   .average()
   .getAsDouble(); // returns 34.4
```

4. Range

We can also create an *IntStream* based on a range:

```
int sum = IntStream.range(1, 10)
   .sum(); // returns 45
int sum = IntStream.rangeClosed(1, 10)
   .sum(); // returns 55
```

As the code snippet above shows there are two ways to create a range of integer values *range()*and *rangeClosed()*.

The difference is that the end of *range()* is exclusive while it is inclusive in *rangeClosed()*.

Range methods are only available for *IntStream* and *LongStream*.

We can use range as a fancy form of a for-each loop:

```
IntStream.rangeClosed(1, 5)
   .forEach(System.out::println);
```

What's good at using them as a for-each loop replacement is that we can also take advantage of the parallel execution:

```
IntStream.rangeClosed(1, 5)
   .parallel()
   .forEach(System.out::println);
```

As helpful as these fancy loops are it's still better to use the traditional for-loops instead of the functional one for simple iterations because of simplicity, readability, and performance in some cases.

5. Boxing and Unboxing

There're times when we need to convert primitive values to their wrapper equivalents.

In those cases, we can use the *boxed()* method:

```
List<Integer> evenInts = IntStream.rangeClosed(1, 10)
   .filter(i -> i % 2 == 0)
   .boxed()
   .collect(Collectors.toList());
```

We can also convert from the wrapper class stream to the primitive stream:

```
// returns 78
int sum = Arrays.asList(33,45)
```

```
3       .stream()
4       .mapToInt(i -> i)
5       .sum();
```

We can always use *mapToXxx* and *flatMapToXxx* methods to create primitive streams.