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()));
```

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

Only when more than 10000 elements. Even then measure first!