Java Fundamentals 3 - Day 1

Overview of the JDK

- Java is not a "bare" programming language
- JDK is rich with classes and features
- "Common" things have been implemented for you already

Things we get for free

- Collections
- Input / Output
- Networking
- BigInteger Mathematics
- GUI Drawing
- Date/Time handling
- Multithreading tools
- JDBC API
- XML Parsing
- Many more …

We can briefly discuss what sort of features we get for free in the JDK. This list is not intended to scare trainees into thinking they will never "understand it all". It is meant as a relief, that you can almost always rely on a piece of the JDK to help you implement your solution. With the confidence that a large number of developers have spent time to make these things work and.

What’s in Java Fundamentals 3

- Java Collections Framework
- Functional Programming in Java
- Stream Processing
- Date / Time API
- Multithreading Basics
- Java Input / Output
- Persistence with JDBC

In this course we try to give you an overview of topics you are most likely to come across during your work. The topics are deep enough to warrant them each their own course. We will make do with giving you insight in the possibilities and where you can find more information.

Why we need Collections!!!

When you think about:

- Group of student
- List of numbers
- A set of mammals types

- A bag contains fruits

Grouping multiple elements into a single unit.

Collection Types

- **Bag**
- **List**
- **Set**
- **Tree**
- **Map or Dictionary**

Bags

- Just put in!
- No Order important
- Can have duplicated objects
- Sometimes called multisets

List

List of Objects that:

- Keep order of insertion
- Can have duplicated objects

Class List

Name				
1.				
2.				
3.				
4.				
5.				
6.				
7.				
8.				
9.				
10.				
11.				
12.				
13.				
14.				
15.				
16.				
17.				
18.				
19.				
20.				
21.				
22.				
23.				
24.				
25.				
26.				
27.				
28.				

Set

- Order is not important
- Duplication is not allowed

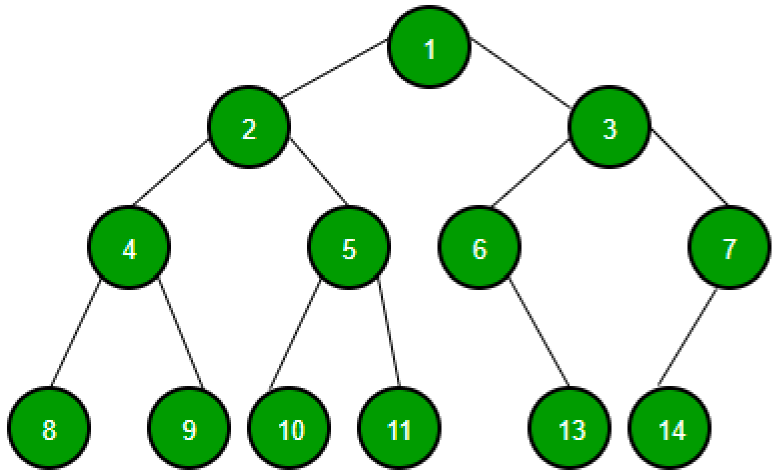
Example

- Set of car brands
- Set of types of shapes

- Set of Odd Numbers

Tree

- Tree like structure
  - Hierarchical data structure
  - One root node
  - One root node with no parent
  - Each node has only one parent and can have multiple children
- No Loop
- Duplicate allowed
- Sorted



When

- fast insertion and retrieval of individual elements
- keep in sorted order.
- Needs fast search on a collection
- Needs to jump between branches
- get ordered a list from a sequence of objects …
- …

Assignment

- Open the project `module-06-collections` from the exercises
- Open the `SimpleListsAssignment` and follow the instructions
- Open the `PerformanceSample` class and run it
- Create an `HashSet` and 2 different Class with same field(s) and implement Hash function in the best and worst way. Then try to inset and retrieve millions of objects and compare the times.e
- Implement a checklist that can be checked or unchecked. Try to change the Comparator in the way that toString prints the Ordered Unchecked items first and then Ordered Checked items.

Sample:

```
[{"checked":false, "task":"Buy a T-shirt"},{"checked": false, "task":"Wash the dishes"},{"checked": false,
"task":"wast the time!"},{"checked": true, "task":"EAT"},{"checked": true, "task":"SLEEP"}]
```

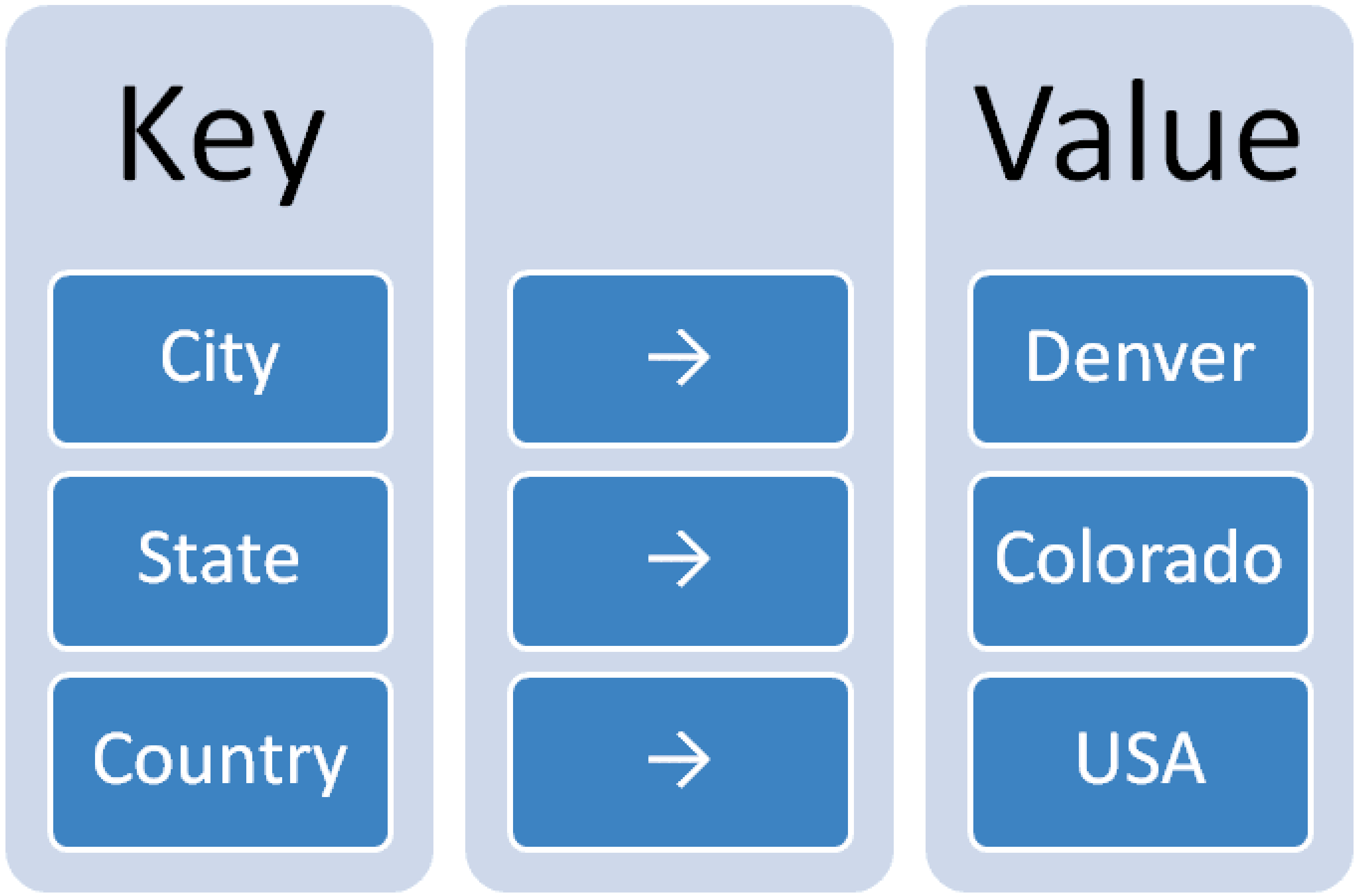
Map

What

- My Object can identify with an ID. Call it Key in Map structure.

!

- As simple as '*KEY ¬ VALUE*'



When

- Fast access to a VALUE by a KEY
- Sort is not important

Collections in Java

Group of individual objects is known as collection

Java Collections Framework: set of interfaces and classes - `java.util` - `java.util.concurrent`

- Two main “root” interfaces:
    - `java.util.Collection`
    - `java.util.Map`
- Type safety
- Bags or multisets should implement this interface directly.
    - implementation: Apache commons and google guava

Advantages

- **Reduces programming effort** by providing data structures and algorithms so you don’t have to write them yourself.
- **Increases performance** by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be tuned by switching implementations.
- **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
- **Reduces the effort required to learn APIs** by requiring you to learn multiple ad hoc collection APIs.
- **Reduces the effort required to design and implement APIs** by not requiring you to produce ad hoc collections APIs.
- **Fosters software reuse** by providing a standard interface for collections and algorithms with which to manipulate them.

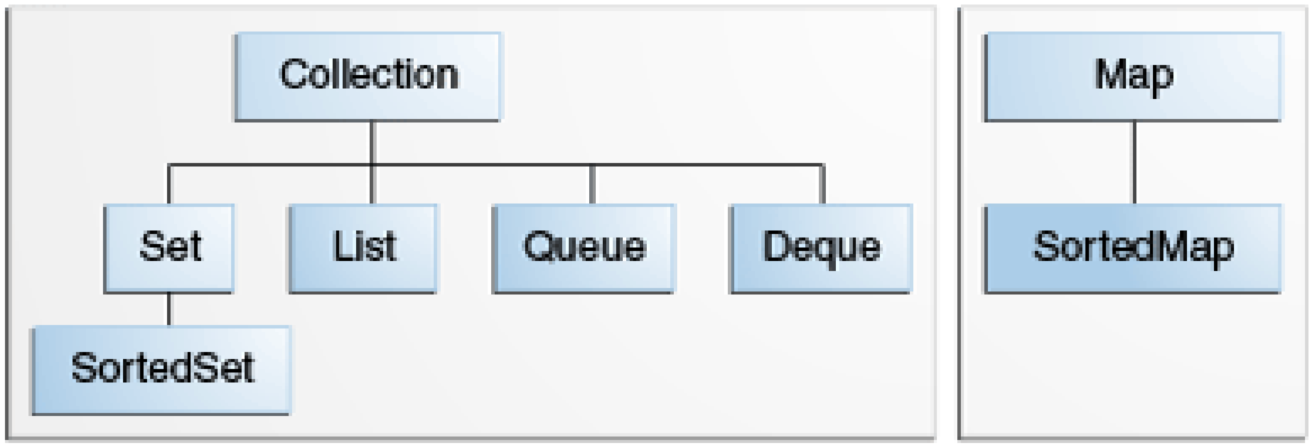
Consists of

- **Collection interfaces**. Represent different types of collections, such as sets, lists, and maps. These interfaces form the basis of the framework.
- **General-purpose implementations**. Primary implementations of the collection interfaces.
- **Legacy implementations**. The collection classes from earlier releases, Vector and Hashtable, were retrofitted to implement the collection interfaces.
- **Special-purpose implementations**. Implementations designed for use in special situations. These implementations display nonstandard performance characteristics, usage restrictions, or behavior.
- **Concurrent implementations**. Implementations designed for highly concurrent use.
- **Wrapper implementations**. Add functionality, such as synchronization, to other implementations.
- **Convenience implementations.** High-performance "mini-implementations" of the collection interfaces.

Consists of (cont')

- **Abstract implementations**. Partial implementations of the collection interfaces to facilitate custom implementations.
- **Algorithms**. Static methods that perform useful functions on collections, such as sorting a list.
- **Infrastructure**. Interfaces that provide essential support for the collection interfaces.
- **Array Utilities.** Utility functions for arrays of primitive types and reference objects. Not, strictly speaking, a part of the collections framework, this feature was added to the Java platform at the same time as the collections framework and relies on some of the same infrastructure.

!



java.util.Collections

- **Static methods** that operate on the Collections or return Collections
- Famous Operations
  - Adding elements to the Collections
  - Sorting a Collection
  - Searching in a Collection
  - Copying Elements
  - Disjoint Collection
- Wrappers The Collections class provides wrapper objects that modify the behavior of standard collections classes in one of three ways
  - synchronizing

    - unmodifiable
    - Checked: checking the type of elements being added to them

Generics in Java Collections

- Generics provide flexible type safety
- Reduce the need for casting
- Move runtime error to compile time
    - compiler keeps track of collection and show error for mismatched type

Generics in Java Collections

- Generic algorithms
    - changing element order in a list
        - reverse, rotate, shuffle, …
    - changing the contents of a list
        - fill, copy, replaceAll
    - finding extreme values in a collection
        - max, min
    - finding specific values in a list
        - binarySearch, indexOfSubList

```
public class CollectionRevolution {

// before generics:
List ints = Arrays.asList( new Integer[] { new Integer(1), new Integer(2) } );

// with generics:
List<Integer> genericInts = Arrays.asList(1,2,3);
}
```

`List`

- Duplicate
- Order of elements
- Positional Access: numerical position
  - `void add(int index, E e)` *add element e at given index*
  - `boolean addAll(int index, Collection<? extends E> c)` *add contents of c at given index*
  - `E get(int index)` *return element with given index*
  - `E remove(int index)` *remove element with given index*
  - `E set(int index, E e)` *replace element with given index by e*

`List` (cont')

- Search
  - `int indexOf(Object o)` *return index of first occurrence of o*

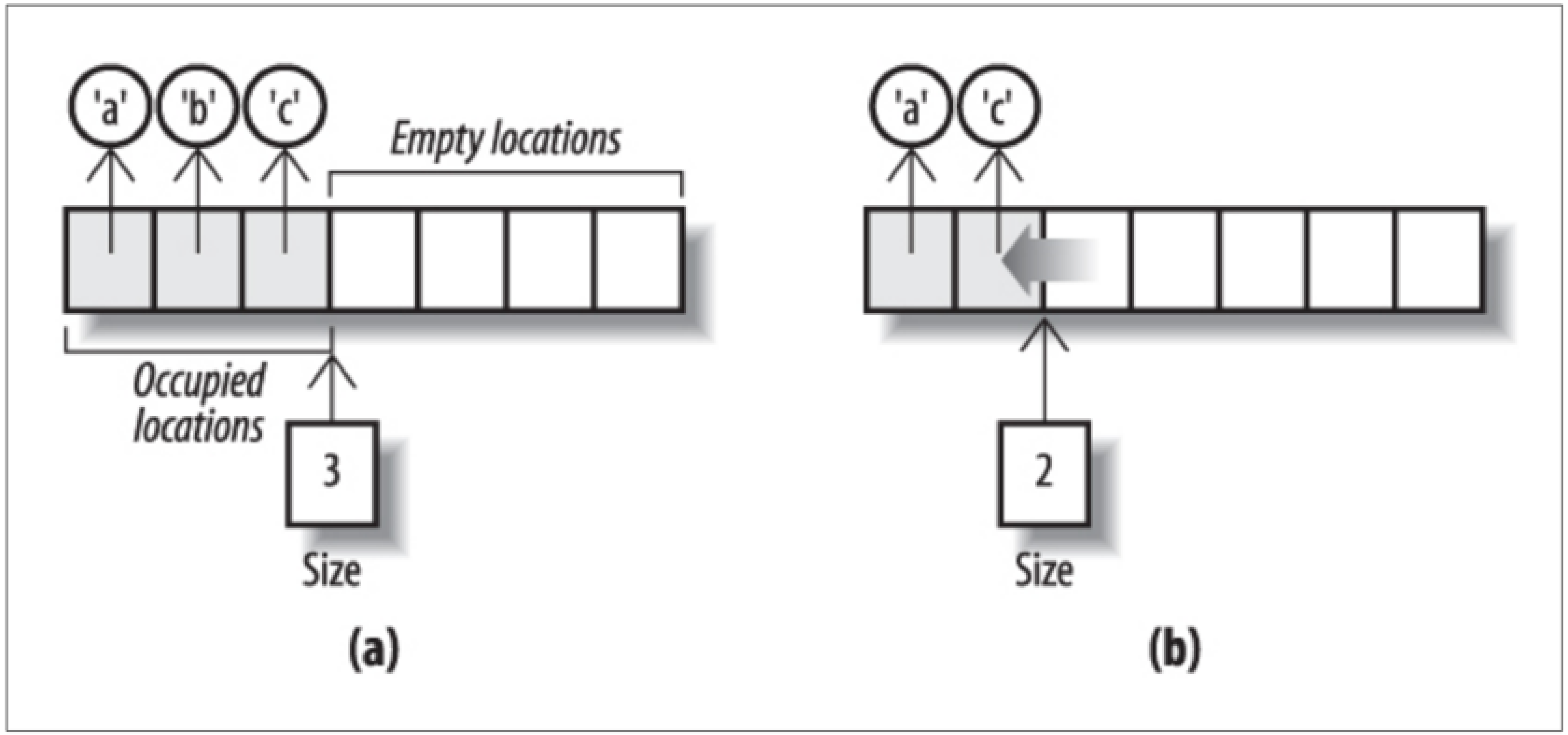
    - `int lastIndexOf(Object o)` *return index of last occurrence of o*

- Range-View
    - `List<E> subList(int fromIndex, int toIndex)` *return a view of a portion of the list*

`ArrayList implements List`

A List backed by an array

- The most commonly used implementation
- once created, they cannot be resized but if an ArrayList has grown to the point where its size is equal to its capacity, attempting to add another element will require it to replace the backing array with a larger one capable of holding the old contents and the new element
- Fast for simple `get` and `set` but for the arbitrary position needs to shift
- Contiguous location on the Memory



`LinkedList implements List`

- NOT at contiguous memory locations
- Dynamic size
- Each element has a pointer to the next element
- Faster than `ArrayList` when removing and adding element (except end of list)
- No random access

- Extra memory for pointer

`LinkedList` implementations

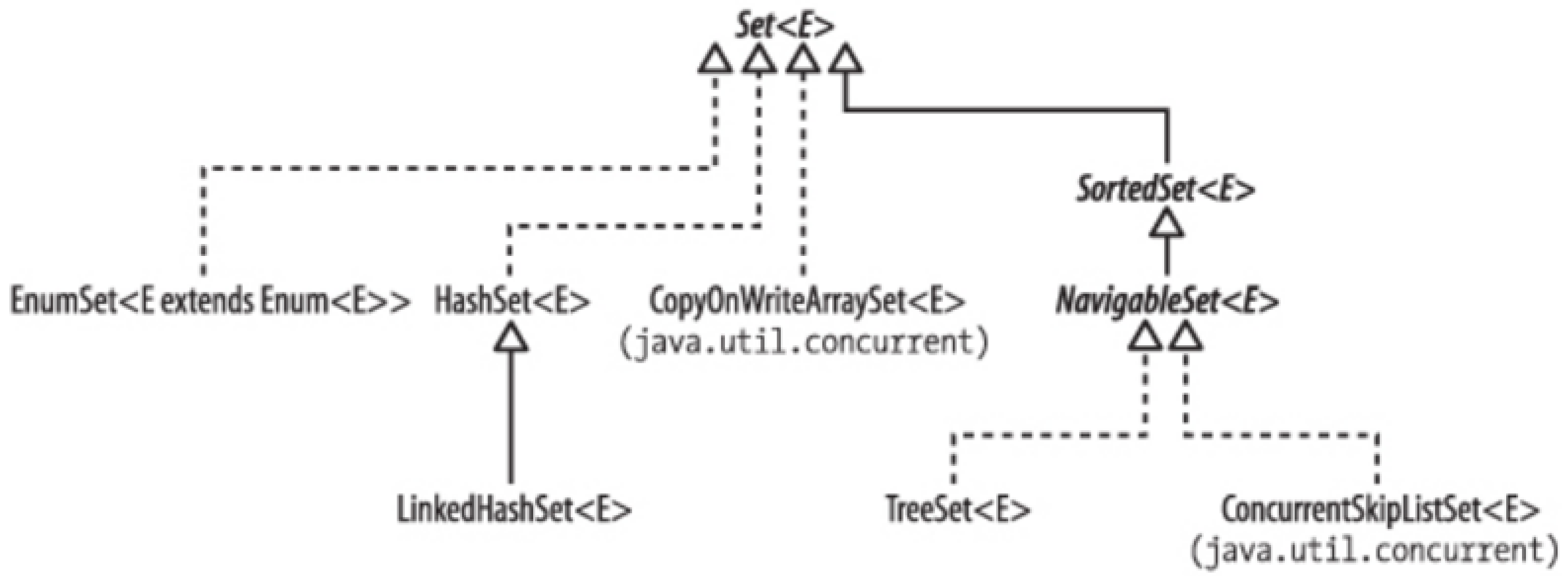
- Singly image::linkedlist.png[]
- Circular image::circular-linkedlist.png[]
- Doubly image::doubly-linkedlist.png[]

Assignment

- Create an `ArrayList` and a `LinkedList` and try to add, remove, set, and find millions of elements then compare the times differences.

`Set`

- No Duplication
- No Order of elements
- 6 implementation in Java core
- Which one? how fast on add, contains, iteration, order



`HashSet implements Set`

- Most commonly
- Use Hash functions and Hash Table
- Fast `set` and `contains`
- Unsychronized and not thread-safe
- `LinkedHashSet`
  - Ordered
  - Improved performance for iteration

`CopyOnWriteArraySet implements Set`

- For lots of Read (O(1)) and a few Write (O(n))
- Thread Safe

`TreeSet implements Set`

- Ordered
  - Needs `Comparable` objects or `Comparator`
- Navigable

Loops on Java Collections

Use for-each

```
import java.util.*;

List<String> arrayStrings = new ArrayList();
 arrayStrings.add("object1");
 arrayStrings.add("object2");
 arrayStrings.add("object3");

 for(String item : arrayStrings){
            System.out.println(item);
 }
```

Use Iterator

```
import java.util.*;

List<String> arrayStrings = new ArrayList();
 arrayStrings.add("object1");
 arrayStrings.add("object2");
 arrayStrings.add("object3");

 Iterator<String> it = arrayStrings.iterator();

 while(it.hasNext()){
      System.out.println(it.next());
 }
```

Use Stream

Create Stream

```
import java.util.*;
import java.util.stream.Stream;

// From List
List<String> arrayStrings = new ArrayList();
 arrayStrings.add("object1");
 arrayStrings.add("object2");
 arrayStrings.add("object3");

 arrayStrings.stream()
 .forEach(item -> System.out.println( item));
```

```
// From Array
String[] stringObjects = new String[] {
    "object 1",
    "object 2",
    "object 3"
};

Stream streamOf = Stream.of(stringObjects);
streamOf.forEach(item -> System.out.println( item));

//Directly from Stream
Stream.of( "object 1",
    "object 2",
    "object 3").forEach(item -> System.out.println( item));

//From Stream builder
Stream.Builder<String> byStreamBuilder = Stream.builder();

byStreamBuilder.accept("object 1");
byStreamBuilder.accept("object 2");
byStreamBuilder.accept("object 3");

byStreamBuilder.build().forEach(item -> System.out.println( item));

//Infinite stream
Stream<Integer> infiniteStream = Stream.iterate(1, i -> i + i);
infiniteStream.limit(10).forEach(item -> System.out.println( item));
```

Stream Operations

- **Terminal operations**: Marks a stream as consumed. Like forEach
- **Intermediate operations**: Locate before Terminal operations and are lazy(Not execute until the result is needed)

forEach

Simplest and most common

map

get a stream and return a stream!

collect

Repackaging elements to a data structure and do additional logics like concatenating

```
import java.util.*;
import java.util.stream.*;

List<String> collectedList = Stream.of("Kevin", "Johannes", "Jeroen", "Thomas", "Tim").map(name -> "Hallo " +
name ).collect(Collectors.toList());

for(String item : collectedList){
            System.out.println(item);
 }
```

filter

get a stream and return a Stream of items that passed the **condition**

```
import java.util.*;
import java.util.stream.*;

```

```
Stream.of(1,2,3,4,5,6,7,8,9,0).filter(num -> num <= 4 )
.forEach(item -> System.out.println( item))
```

Lazy Evaluation

null value will not process in blow example.

```
import java.util.*;
import java.util.stream.*;

System.out.println(
        Stream.of(1,2,3,4,null,5).filter(number -> 2/number==1 ).findFirst().get());
```

toArray

Collect the stream as an array

```
import java.util.*;
import java.util.stream.*;

Integer[] filteredArray=Stream.of(1,2,3,4,5,6,7,8,9,0).filter(num -> num <= 4 ).toArray(Integer[]::new);

for(int i=0;i<filteredArray.length;i++){
    System.out.println(filteredArray[i]);
}
```

flatMap

Use to flatten the steam of List objects like Stream<List<String>>

```
import java.util.*;
import java.util.stream.*;

List<List<String>> nestedList = Arrays.asList(
      Arrays.asList("Kevin", "Johannes", "Tim"),
      Arrays.asList( "Jeroen", "Thomas"),
      Arrays.asList("Hans", "Tom"));

List<String> flaternList = nestedList.stream().flatMap(Collection::stream).collect(Collectors.toList());

for(String name: flaternList){
    System.out.println(name);
}
```

peek

An intermediate operation that is used to apply multiple operation on each item of stream.

```
import java.util.*;
import java.util.stream.*;

Stream.of(1,-2,3,4,5,-6,7,8,9,0)
.peek(item -> Math.abs(item) )
.peek(item ->  System.out.println(item) )
.map(item -> Math.abs(item) )
.forEach(item -> System.out.println(item) );
```

Comparison based operations

- sorted
- min
- max
- distinct
- allMatch
- anyMatch
- noneMatch

```
import java.util.*;
import java.util.stream.*;
import java.util.function.Supplier;

Supplier<Stream<Integer>> streamSupplier = () ->   Stream.of(4,-2,3,3,1,5,-6,7,8,9);

System.out.println(streamSupplier.get().sorted((i,j) ->
i.compareTo(j)).map(String::valueOf).collect(Collectors.joining(",")));

System.out.println(streamSupplier.get().min(Integer::compareTo).get());

System.out.println(streamSupplier.get().max(Integer::compareTo).get());

System.out.println(streamSupplier.get().distinct().map(String::valueOf).collect(Collectors.joining(",")));


System.out.println(streamSupplier.get().allMatch(i -> i > 0));

System.out.println(streamSupplier.get().anyMatch(i -> i < 0));
System.out.println(streamSupplier.get().noneMatch(i -> i > 0));
```

Operations on Numbers

- sum
- average
- range

```
import java.util.*;
import java.util.stream.*;
import java.util.function.Supplier;

Supplier<Stream<Integer>> streamSupplier = () ->  Stream.of(4,-2,3,3,1,5,-6,7,8,9);

System.out.println("Average:" +
                                streamSupplier.get().mapToInt(number -> number).average().getAsDouble());


System.out.println("Sum:" +
                            streamSupplier.get().mapToInt(number -> number).sum());
```

Reduction Operations

To process the stream and get a result. Like sum, average

```
import java.util.*;
import java.util.stream.*;
```

```
Stream<Integer> numnerStream = Stream.of(4,-2,3,3,1,5,-6,7,8,9);

 System.out.println("Sum:" +
                numnerStream.reduce(0, Integer::sum));


 Stream<Integer> numnerStream2 = Stream.of(1,2,3);

        System.out.println("Sum:" +
                numnerStream2.reduce(0.0,(result,number)-> result + Math.pow(2,number),(result1, result2)
->result1+result2 ));
```

Advanced Operations

- joining
- toSet
- toCollection
- summarizingDouble
- partitioningBy
- groupingBy
- mapping
- reducing

Works with Queue

```
import java.util.*;
```

```
Deque<String> myQueue = new LinkedList<String>();

myQueue.add("object1");
myQueue.add("object2");
myQueue.add("object3");

//adds "hight priority" to start of myQueue
myQueue.addFirst("hight priority");


//adds "low priority" to end of myQueue
myQueue.offerLast("low priority");

// print first element
System.out.println( myQueue.peekFirst());

// print last element
System.out.println( myQueue.peekLast());

while(myQueue.size()>0){
    System.out.println( myQueue.poll());
}
```

Iterate over a Map

```
import java.util.*;

Map<Integer,String> objectMap  = new HashMap<>();

objectMap.put(1,"object 1");
objectMap.put(2,"object 1");
objectMap.put(3,"object 2");
objectMap.put(4,"object 3");

//Print All keys
System.out.println("Print all keys");
 objectMap.keySet().stream()
 .forEach(item -> System.out.println( item));

//Print All keys
System.out.println("Print all values");
 objectMap.values().stream()
 .forEach(item -> System.out.println( item));


//Print All keys
System.out.println("Print all values");
 objectMap.entrySet().stream()
 .forEach(entrySet -> System.out.println( entrySet.getKey() +" ==> "+entrySet.getValue()));
```

Assignment

Define a Student class with name, birth year and the number of passed exams.

Create a collection of students and find the list of students with 20 passed exams and sort with name and print on console.