Java and Functional Programing:

Creating a Stream

```
// Create an ArrayList
List<Integer> myList = new ArrayList<Integer>();
myList.add(1);
myList.add(5);
myList.add(8);

// Convert it into a Stream
Stream<Integer> myStream = myList.stream();
```

```
// Create an array
Integer[] myArray = {1, 5, 8};

// Convert it into a Stream
Stream<Integer> myStream = Arrays.stream(myArray);
```

The `map` Method

Once you have a `Stream` object, you can use a variety of methods to transform it into another `Stream` object. The first such method we're going to look at is the `map` method. It takes a lambda expression as its only argument, and uses it to change every individual element in the stream. Its return value is a new `Stream` object containing the changed elements.

```
String[] myArray = new String[]{"bob", "alice", "paul", "ellie"};
Stream<String> myStream = Arrays.stream(myArray);
```

```
Stream<String> myNewStream =
            myStream.map(s -> s.toUpperCase());
```

The Stream object returned contains the changed strings. To convert it into an array, you use its toArray method:

```
String[] myNewArray =
            myNewStream.toArray(String[]::new);
```

The `filter` Method

In the previous section, you saw that the `map` method processes every single element in a `Stream` object. You might not always want that. Sometimes, you might want to work with only a subset of the elements. To do so, you can use the `filter` method.

Just like the `map` method, the `filter` method expects a lambda expression as its argument. However, the lambda expression passed to it must always return a `boolean` value, which determines whether or not the processed element should belong to the resulting `Stream` object.

```
Arrays.stream(myArray)
      .filter(s -> s.length() > 4)
      .toArray(String[]::new);
```

Reduction Operations

A reduction operation is one which allows you to compute a result using all the elements present in a stream. Reduction operations are also called terminal operations because they are always

present at the end of a chain of `Stream` methods. We've already been using a reduction method in our previous examples: the `toArray` method. It's a terminal operation because it converts a `Stream` object into an array.

Java 8 includes several reduction methods, such as `sum`, `average` and `count`, which allow to perform arithmetic operations on `Stream` objects and get numbers as results. For example, if you want to find the sum of an array of integers, you can use the following code:

```
int myArray[] = { 1, 5, 8 };
int sum = Arrays.stream(myArray).sum();
```

If you want to perform more complex reduction operations, however, you must use the `reduce` method. Unlike the `map` and `filter` methods, the `reduce` method expects two arguments: an identity element, and a lambda expression. You can think of the identity element as an element which does not alter the result of the reduction operation. For example, if you are trying to find the product of all the elements in a stream of numbers, the identity element would be the number 1.

The lambda expression you pass to the `reduce` method must be capable of handling two inputs: a partial result of the reduction operation, and the current element of the stream. If you are wondering what a partial result is, it's the result obtained after processing all the elements of the stream that came before the current element.

The following is a sample code snippet which uses the `reduce` method to concatenate all the elements in an array of `String` objects:

```
String[] myArray = { "this", "is", "a", "sentence" };
String result = Arrays.stream(myArray)
                .reduce("", (a,b) -> a + b);
```

Java 8 Streams

1) What Are Streams?

Streams can be defined as a sequences of elements from a source which support data processing operations. You can treat streams as operations on data. You will get to know as you go through this article.

2) Why Streams?

Almost every Java application use Collections API to store and process the data. Despite being the most used Java API, it is not easy to write the code for even some common data processing operations like filtering, finding, matching, sorting, mapping etc using Collections API . So, there needed Next-Gen API to process the data. So Java API designers have come with Java 8 Streams API to write more complex data processing operations with much of ease.

3) Characteristics Of Java 8 Streams

3.1) Streams are not the data structures

Streams doesn't store the data. You can't add or remove elements from streams. Hence, they are not the data structures. They are the just operations on data.

3.2) Stream Consumes a data source

Stream consumes a source, performs operations on it and produces the result. Source may be a collection or an array or an I/O resource. Remember, stream doesn't modify the source.

3.3) Intermediate And Terminal Operations

Most of the stream operations return another new stream and they can be chained together to form a pipeline of operations.

The operations which return stream themselves are called intermediate operations. For example – ***filter()***, ***distinct()***, ***sorted()*** **etc.**

The operations which return other than stream are called terminal operations. ***count()***. ***min()***, ***max()*** are some terminal operations.

3.4) Pipeline Of Operations

A pipeline of operations consists of three things – a source, one or more intermediate operations and a terminal operation. Pipe-lining of operations let you to write database-like queries on a data source. In the below example, int array is the source, *filter()* and *distinct()* are intermediate operations and *forEach()* is a terminal operation.

```
IntStream.of(new int[] {4, 7, 1, 8, 3, 9, 7}).filter((int i) -> i >
5).distinct().forEach(System.out::println);
```

3.5) Internal Iteration

Collections need to be iterated explicitly. i.e you have to write the code to iterate over collections. But, all stream operations do the iteration internally behind the scene for you. You need not to worry about iteration at all while writing the code using Java 8 Streams API.

3.6) Parallel Execution

To gain the performance while processing the large amount of data, you have to process it in parallel and use multi core architectures. Java 8 Streams can be processed in parallel without writing any multi threaded code. For example, to process the collections in parallel, you just use *parallelStream()* method instead of *stream()* method.

```
List<String> names = new ArrayList<>();
names.add("David");
names.add("Johnson");
names.add("Samontika");
names.add("Brijesh");
names.add("John");

//Normal Execution

names.stream().filter((String name) -> name.length() >
5).skip(2).forEach(System.out::println);

//Parallel Execution

names.parallelStream().filter((String name) -> name.length() >
5).skip(2).forEach(System.out::println);
```

3.7) Streams are lazily populated

All elements of a stream are not populated at a time. They are lazily populated as per demand because intermediate operations are not evaluated until terminal operation is invoked.

3.8) Streams are traversable only once

You can't traverse the streams more than once just like iterators. If you traverse the stream first time, it is said to be consumed.

```
List<String> nameList = Arrays.asList("Dinesh", "Ross", "Kagiso", "Steyn");

Stream<String> stream = nameList.stream();
stream.forEach(System.out::println);
stream.forEach(System.out::println);

//Error : stream has already been operated upon or closed
```

3.9) Short Circuiting Operations

Short circuiting operations are the operations which don't need the whole stream to be processed to produce a result. For example – *findFirst()*, *findAny()*, *limit()* etc.

4) java.util.stream.Stream

java.util.stream.Stream interface is the center of Java 8 Streams API. This interface contains all the stream operations. Below table shows frequently used *Stream* methods with description.

Operation	Method Signature	Type Of Operation	What It Does?
filter()	Stream<T> filter(Predicate<T> predicate)	Intermediate	Returns a stream of elements which satisfy the given predicate.
map()	Stream<R> map(Function< T, R> mapper)	Intermediate	Returns a stream consisting of results after applying given function to elements of the stream.
distinct()	Stream<T> distinct()	Intermediate	Returns a stream of unique elements.
sorted()	Stream<T> sorted()	Intermediate	Returns a stream consisting of elements sorted according to natural order.
limit()	Stream<T> limit(long maxSize)	Intermediate	Returns a stream containing first *n* elements.
skip()	Stream<T> skip(long n)	Intermediate	Returns a stream after skipping first *n* elements.
forEach()	void forEach(Consumer<T> action)	Terminal	Performs an action on all elements of a stream.
toArray()	Object[] toArray()	Terminal	Returns an array containing elements of a stream.
reduce()	T reduce(T identity, BinaryOperator<T> accumulator)	Terminal	Performs reduction operation on elements of a stream using initial value and binary operation.
collect()	R collect(Collector<T> collector)	Terminal	Returns mutable result container such as List or Set.
min()	Optional<T> min(Comparator<T> comparator)	Terminal	Returns minimum element in a stream wrapped in an Optional object.
max()	Optional<T> max(Comparator<T> comparator)	Terminal	Returns maximum element in a stream wrapped in an Optional object.
count()	long count()	Terminal	Returns the number of elements in a stream.
anyMatch()	boolean anyMatch(Predicate<T> predicate)	Terminal	Returns true if any one element of a stream matches with given predicate.
allMatch()	boolean allMatch(Predicate<T> predicate)	Terminal	Returns true if all the elements of a stream matches with given predicate.
noneMatch()	boolean noneMatch(Predicate< T> predicate)	Terminal	Returns true only if all the elements of a stream doesn't match with given predicate.
findFirst()	Optional<T> findFirst()	Terminal	Returns first element of a stream wrapped in an Optional object.
findAny()	Optional<T> findAny()	Terminal	Randomly returns any one element in a stream.

Let's see some important stream operations with examples.

5) Java 8 Stream Operations

5.1) Stream Creation Operations

5.1.1) *empty()* : Creates an empty stream

Method Signature: public static<T> Stream<T> empty()
Type of Method: Static Method
What It Does? Returns an empty stream of type T.

```
Stream<Student> emptyStream = Stream.empty();

System.out.println(emptyStream.count());

//Output: 0
```

5.1.2) *of(T t)* : Creates a stream of single element of type T

Method Signature: public static<T> Stream<T> of(T t)
Type of Method: Static Method
What It Does? Returns a single element stream of type T.

```
Stream<Student> singleElementStream = Stream.of(new Student());

System.out.println(singleElementStream.count());

//Output: 1
```

5.1.3) *of(T… values)* : Creates a stream from values

Method Signature: public static<T> Stream<T> of(T… values)
Type of Method: Static Method
What It does? Returns a stream consisting of supplied values as elements.

```
Stream<Integer> streamOfNumbers = Stream.of(7, 2, 6, 9, 4, 3, 1);

System.out.println(streamOfNumbers.count());

//Output: 7
```

5.1.4) Creating streams from collections

From Java 8, every collection type will have a method called *stream()* which returns the stream of respective collection type.

Example: Creating a stream from List

```
List<String> listOfStrings = new ArrayList<>();
listOfStrings.add("One");
listOfStrings.add("Two");
listOfStrings.add("Three");
listOfStrings.stream().forEach(System.out::println);

// Output:
```

```
// One
// Two
// Three
```

5.2) Selection Operations

5.2.1) *filter()* : Selecting with a predicate

Method Signature: Stream<T> filter(Predicate<T> predicate)
Type of Operation: Intermediate Operation
What it does? Returns a stream of elements which satisfy the given predicate.

```
List<String> names = new ArrayList<>();

names.add("David");
names.add("Johnson");
names.add("Samontika");
names.add("Brijesh");
names.add("John");

//Selecting names containing more than 5 characters

names.stream().filter((String name) -> name.length() > 5).forEach(System.out::println);

// Output:
// Johnson
// Samontika
// Brijesh
```

5.2.2) *distinct()* : Selects only unique elements

Method Signature: Stream<T> distinct()
Type of Operation: Intermediate Operation
What It Does? Returns a stream of unique elements.

```
List<String> names = new ArrayList<>();

names.add("David");
names.add("Johnson");
names.add("Samontika");
names.add("Brijesh");
names.add("John");
names.add("David");
names.add("Brijesh");

//Selecting only unique names

names.stream().distinct().forEach(System.out::println);

// Output:
// David
// Johnson
// Samontika
// Brijesh
// John
```

5.2.3) *limit()* : Selects first *n* elements

Method Signature: Stream<T> limit(long maxSize)
Type of Operation: Intermediate Operation

What It Does? Returns a stream containing first n elements.

```
List<String> names = new ArrayList<>();

names.add("David");
names.add("Johnson");
names.add("Samontika");
names.add("Brijesh");
names.add("John");
names.add("David");
names.add("Brijesh");

//Selecting first 4 names

names.stream().limit(4).forEach(System.out::println);

// Output:

// David
// Johnson
// Samontika
// Brijesh
```

5.2.4) *skip()* : Skips first n elements

Method Signature: **Stream**<T> skip(long n)
Type of Operation: Intermediate Operation
What It Does? Returns a stream after skipping first n elements.

```
List<String> names = new ArrayList<>();

names.add("David");
names.add("Johnson");
names.add("Samontika");
names.add("Brijesh");
names.add("John");
names.add("David");
names.add("Brijesh");

//Skipping first 4 names

names.stream().skip(4).forEach(System.out::println);

// Output:

// John
// David
// Brijesh
```

5.3) Mapping Operations

5.3.1) *map()* : Applies a function

Method Signature: Stream<R> map(Function<T, R> mapper);
Type of Operation: Intermediate Operation
What It Does? Returns a stream consisting of results after applying given function to elements of the stream.

```
List<String> names = new ArrayList<>();

names.add("David");
names.add("Johnson");
names.add("Samontika");
names.add("Brijesh");
names.add("John");

//Returns length of each name
                names.stream().map(String::length).forEach(System.out::println);

// Output:

// 5
// 7
// 9
// 7
// 4
```

Other versions of *map()* method : *mapToInt()*, *mapToLong()* and *mapToDouble()*.

5.4) Sorting Operations

5.4.1) *sorted()* : Sorting according to natural order

Method Signature: Stream<T> sorted()
Type of Operation: Intermediate Operation
What It Does? Returns a stream consisting of elements sorted according to natural order.

```
List<String> names = new ArrayList<>();

names.add("David");
names.add("Johnson");
names.add("Samontika");
names.add("Brijesh");
names.add("John");

//Sorting the names according to natural order

names.stream().sorted().forEach(System.out::println);

// Output:
// Brijesh
// David
// John
// Johnson
// Samontika
```

5.4.2) *sorted(Comparator)* : Sorting according to supplied comparator

Method Signature: Stream<T> sorted(Comparator<T> comparator)
Type of Operation: Intermediate Operation
What It Does: Returns a stream consisting of elements sorted according to supplied Comparator.

```
List<String> names = new ArrayList<>();

names.add("David");
names.add("Johnson");
```

```
names.add("Samontika");
names.add("Brijesh");
names.add("John");

//Sorting the names according to their length

names.stream().sorted((String name1, String name2) -> name1.length() -
name2.length()).forEach(System.out::println);

// Output:

// John
// David
// Johnson
// Brijesh
// Samontika
```

5.5) Reducing Operations

Reducing operations are the operations which combine all the elements of a stream repeatedly to produce a single value. For example, counting number of elements, calculating average of elements, finding maximum or minimum of elements etc.

5.5.1) *reduce()* : Produces a single value

Method Signature: T reduce(T identity, BinaryOperator<T> accumulator);
Type of Operation: Terminal Operation
What It Does? This method performs reduction operation on elements of a stream using initial value and binary operation.

```
int sum = Arrays.stream(new int[] {7, 5, 9, 2, 8, 1}).reduce(0, (a, b) -> a+b);
//Output : 32
```

There is another form of *reduce()* method which takes no initial value. But returns an *Optional* object.

```
OptionalInt sum = Arrays.stream(new int[] {7, 5, 9, 2, 8, 1}).reduce((a, b) ->
a+b);
//Output : OptionalInt[32]
```

Methods *min()*, *max()*, *count()* and *collect()* are special cases of reduction operation.

5.5.2) *min()* : Finding the minimum

Method Signature: Optional<T> min(Comparator<T> comparator)
Type of Operation: Terminal Operation
What It Does? It returns minimum element in a stream wrapped in an Optional object.

```
OptionalInt min = Arrays.stream(new int[] {7, 5, 9, 2, 8, 1}).min();
//Output : OptionalInt[1]
//Here, min() of IntStream will be used as we are passing an array of ints
```

5.5.3) *max()* : Finding the maximum

Method Signature: Optional<T> max(Comparator<T> comparator)
Type of Operation: Terminal Operation
What It Does? It returns maximum element in a stream wrapped in an Optional object.

```
OptionalInt max = Arrays.stream(new int[] {7, 5, 9, 2, 8, 1}).max();
//Output : OptionalInt[9]

//Here, max() of IntStream will be used as we are passing an array of ints
```

5.5.4) *count()* : Counting the elements

Method Signature: long count()
Type of Operation: Terminal Operation
What It Does? Returns the number of elements in a stream.

```
List<String> names = new ArrayList<>();

names.add("David");
names.add("Johnson");
names.add("Samontika");
names.add("Brijesh");
names.add("John");

//Counting the names with length > 5

long noOfBigNames = names.stream().filter((String name) -> name.length() > 5).count();

System.out.println(noOfBigNames);

//  Output : 3
```

5.5.5) *collect()* : Returns mutable container

Method Signature: R collect(Collector<T> collector)
Type of Operation: Terminal Operation
What It Does? *collect()* method is a special case of reduction operation called mutable reduction operation because it returns mutable result container such as List or Set.

```
List<String> names = new ArrayList<>();

names.add("David");
names.add("Johnson");
names.add("Samontika");
names.add("Brijesh");
names.add("John");

//Storing first 3 names in a mutable container

List<String> first3Names =
names.stream().limit(3).collect(Collectors.toList());
System.out.println(first3Names);

//  Output : [David, Johnson, Samontika]
```

5.6) Finding And Matching Operations

5.6.1) *anyMatch()* : Any one element matches

Method Signature: boolean anyMatch(Predicate<T> predicate)
Type of Operation: Short-circuiting Terminal Operation

What It Does? Returns true if any one element of a stream matches with given predicate. This method may not evaluate all the elements of a stream. Even if the first element matches with given predicate, it ends the operation.

```
List<String> names = new ArrayList<>();

names.add("David");
names.add("Johnson");
names.add("Samontika");
names.add("Brijesh");
names.add("John");

if(names.stream().anyMatch((String name) -> name.length() == 5))
{
    System.out.println("Yes... There is a name exist with 5 letters");
}
```

5.6.2) *allMatch()* : All elements matches

Method Signature: boolean allMatch(Predicate<T> predicate)
Type of Operation: Terminal Operation
What It Does? This method returns true if all the elements of a stream matches with given predicate. Otherwise returns false.

```
List<String> names = new ArrayList<>();

names.add("Sampada");
names.add("Johnson");
names.add("Samontika");
names.add("Brijesh");

if(names.stream().allMatch((String name) -> name.length() > 5))
{
    System.out.println("All are big names");
}
```

5.6.3) *noneMatch()* : No element matches

Method Signature: boolean noneMatch(Predicate<T> predicate)
Type of Operation: Terminal Operation
What It Does? Returns true only if all the elements of a stream doesn't match with given predicate.

```
List<String> names = new ArrayList<>();

names.add("David");
names.add("Johnson");
names.add("Samontika");
names.add("Brijesh");
names.add("John");

if(names.stream().noneMatch((String name) -> name.length() == 2))
{
    System.out.println("There is no two letter name");
}
```

5.6.4) *findFirst()* : Finding first element

Method Signature: Optional<T> findFirst()
Type of Operation: Short-circuiting Terminal Operation

What It Does? Returns first element of a stream wrapped in an *Optional* object.

```
Optional<String> firstElement = Stream.of("First", "Second", "Third",
"Fourth").findFirst();

//Output : Optional[First]
```

5.6.5) *findAny()* : Finding any element

Method Signature: Optional<T> findAny()
Type of Operation: Short-circuiting Terminal operation
What It Does? Randomly returns any one element in a stream. The result of this operation is unpredictable. It may select any element in a stream. Multiple invocations on the same source may not return same result.

```
Optional<String> anyElement = Stream.of("First", "Second", "Third",
"Fourth").findAny();
```

5.7) Other Operations

5.7.1) *forEach()* :

Method Signature: void forEach(Consumer<T> action)
Type of Operation: Terminal Operation
What It Does? Performs an action on all elements of a stream.

```
Stream.of("First", "Second", "Second", "Third",
"Fourth").limit(3).distinct().forEach(System.out::println);

// Output

// First
// Second
```

5.7.2) *toArray()* : Stream to array

Method Signature: Object[] toArray()
Type of Operation: Terminal Operation
What It Does? Returns an array containing elements of a stream.

```
List<String> names = new ArrayList<>();

names.add("David");
names.add("Johnson");
names.add("Samontika");
names.add("Brijesh");
names.add("John");

//Storing first 3 names in an array

Object[] streamArray = names.stream().limit(3).toArray();
System.out.println(Arrays.toString(streamArray));

// Output

// [David, Johnson, Samontika]
```

5.7.3) *peek()* :

Method Signature: Stream<T> peek(Consumer<T> action)
Type of Operation: Intermediate Operation
What It Does? Performs an additional action on each element of a stream. This method is only to support debugging where you want to see the elements as you pass in a pipeline.

```
List<String> names = new ArrayList<>();

names.add("David");
names.add("Johnson");
names.add("Samontika");
names.add("Brijesh");
names.add("John");
names.add("David");

names.stream()
        .filter(name -> name.length() > 5)
        .peek(e -> System.out.println("Filtered Name :"+e))
        .map(String::toUpperCase)
        .peek(e -> System.out.println("Mapped Name :"+e))
        .toArray();

//Output:

//Filtered Name :Johnson
//Mapped Name :JOHNSON
//Filtered Name :Samontika
//Mapped Name :SAMONTIKA
//Filtered Name :Brijesh
//Mapped Name :BRIJESH
```

Java 8 Collectors Tutorial

Java 8 Collectors tutorial mainly consist of three things – *Stream.collect()* method, *Collector* interface and *Collectors* class. *collect()* method is a terminal operation in *Stream* interface. *Collector* is an interface in *java.util.stream* package. *Collectors* class, also a member of *java.util.stream* package, is an utility class containing many static methods which perform some common reduction operations.

1) Stream.collect() *Method*

collect() method is a terminal operation in *Stream* interface. It is a special case of reduction operation called mutable reduction operation because it returns mutable result container such as *List*, *Set* or *Map* according to supplied *Collector*.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class CollectorsExamples
{
    public static void main(String[] args)
    {
        List<Integer> numbers = Arrays.asList(8, 2, 5, 7, 3, 6);
        //collect() method returning List of OddNumbers
```

```
        List<Integer> oddNumbers = numbers.stream().filter(i -> i%2 !=
0).collect(Collectors.toList());

        System.out.println(oddNumbers);

        //OUTPUT : [5, 7, 3]
    }
}
```

2) java.util.stream.Collector *Interface*

java.util.stream.Collector interface contains four functions that work together to accumulate input elements into a mutable result container and optionally performs a final transformation on the result. Those four functions are,

a) *Supplier()* :

A function that creates and returns a new mutable result container.

b) *accumulator()* :

A function that accumulates a value into a mutable result container.

c) *combiner()* :

A function that accepts two partial results and merges them.

d) *finisher()* :

A function that performs final transformation from the intermediate accumulation type to the final result type.

3) java.util.stream.Collectors *Class*

java.util.stream.Collectors class contains static factory methods which perform some common reduction operations such as accumulating elements into Collection, finding min, max, average, sum of elements etc. All the methods of *Collectors* class return *Collector* type which will be supplied to *collect()* method as an argument.

Let's see *Collectors* class methods one by one.
In the below coding examples, we will be using following *Student* class and *studentList*.

Student Class :

```
class Student
{
    String name;

    int id;

    String subject;

    double percentage;

    public Student(String name, int id, String subject, double percentage)
    {
        this.name = name;
```

```
        this.id = id;
        this.subject = subject;
        this.percentage = percentage;
    }

    public String getName()
    {
        return name;
    }

    public int getId()
    {
        return id;
    }

    public String getSubject()
    {
        return subject;
    }

    public double getPercentage()
    {
        return percentage;
    }

    @Override
    public String toString()
    {
        return name+"-"+id+"-"+subject+"-"+percentage;
    }
}
```

studentList **:**

```
List<Student> studentList = new ArrayList<Student>();

studentList.add(new Student("Paul", 11, "Economics", 78.9));
studentList.add(new Student("Zevin", 12, "Computer Science", 91.2));
studentList.add(new Student("Harish", 13, "History", 83.7));
studentList.add(new Student("Xiano", 14, "Literature", 71.5));
studentList.add(new Student("Soumya", 15, "Economics", 77.5));
studentList.add(new Student("Asif", 16, "Mathematics", 89.4));
studentList.add(new Student("Nihira", 17, "Computer Science", 84.6));
studentList.add(new Student("Mitshu", 18, "History", 73.5));
studentList.add(new Student("Vijay", 19, "Mathematics", 92.8));
studentList.add(new Student("Harry", 20, "History", 71.9));
```

3.1) *Collectors.toList()* :

It returns a *Collector* which collects all input elements into a new *List*.

Example : Collecting top 3 performing students into *List*

```
List<Student> top3Students =
studentList.stream().sorted(Comparator.comparingDouble(Student::getPercentage).reversed
()).limit(3).collect(Collectors.toList());

System.out.println(top3Students);

//Output :

//[Vijay-19-Mathematics-92.8, Zevin-12-Computer Science-91.2, Asif-16-Mathematics-89.4]
```

3.2) *Collectors.toSet()* :

It returns a *Collector* which collects all input elements into a new *Set*.

Example : Collecting subjects offered into *Set*.

```
Set<String> subjects =
studentList.stream().map(Student::getSubject).collect(Collectors.toSet());

System.out.println(subjects);

//Output :

//[Economics, Literature, Computer Science, Mathematics, History]
```

3.3) *Collectors.toMap()* :

This method returns a *Collector* which collects input elements into a *Map* whose keys and values are the result of applying mapping functions to input elements.

Example : Collecting name and percentage of each student into a *Map*

```
Map<String, Double> namePercentageMap =
studentList.stream().collect(Collectors.toMap(Student::getName,
Student::getPercentage));

System.out.println(namePercentageMap);

//Output :

//{Asif=89.4, Vijay=92.8, Zevin=91.2, Harry=71.9, Xiano=71.5, Nihira=84.6, Soumya=77.5,
Mitshu=73.5, Harish=83.7, Paul=78.9}
```

3.4) *Collectors.toCollection()* :

This method returns a *Collector* which collects all input elements into a new *Collection*.

Example : Collecting first 3 students into *LinkedList*

```
LinkedList<Student> studentLinkedList =
studentList.stream().limit(3).collect(Collectors.toCollection(LinkedList::new));

System.out.println(studentLinkedList);

//Output :

//[Paul-11-Economics-78.9, Zevin-12-Computer Science-91.2, Harish-13-History-83.7]
```

3.5) *Collectors.joining()* :

This method returns a *Collector* which concatenates input elements separated by the specified delimiter.

Example : Collecting the names of all students joined as a string

```
String namesJoined =
studentList.stream().map(Student::getName).collect(Collectors.joining(", "));

System.out.println(namesJoined);

//Output :

//Paul, Zevin, Harish, Xiano, Soumya, Asif, Nihira, Mitshu, Vijay, Harry
```

3.6) *Collectors.counting()* :

It returns a *Collector* that counts number of input elements.

Example : Counting number of students.

```
Long studentCount = studentList.stream().collect(Collectors.counting());

System.out.println(studentCount);

//Output : 10
```

3.7) *Collectors.maxBy()* :

This method returns a *Collector* that collects largest element in a stream according to supplied *Comparator*.

Example : Collecting highest percentage.

This method returns a *Collector* that collects largest element in a stream according to supplied *Comparator*.

Example : Collecting highest percentage.

```
Optional<Double> highPercentage =
studentList.stream().map(Student::getPercentage).collect(Collectors.maxBy(Comparator.na
turalOrder()));

System.out.println(highPercentage);

//Output : Optional[92.8]
```

3.8) *Collectors.minBy()* :

This method returns a *Collector* which collects smallest element in a stream according to supplied *Comparator*.

Example : Collecting lowest percentage.

```
Optional<Double> lowPercentage =
studentList.stream().map(Student::getPercentage).collect(Collectors.minBy(Comparator.na
turalOrder()));

System.out.println(lowPercentage);

//Output : Optional[71.5]
```

3.9) *summingInt(), summingLong(), summingDouble()*

These methods returns a *Collector* which collects sum of all input elements.

Example : Collecting sum of percentages

```
Double sumOfPercentages =
studentList.stream().collect(Collectors.summingDouble(Student::getPercentage));

System.out.println(sumOfPercentages);

//Output : 815.0
```

3.10) *averagingInt(), averagingLong(), averagingDouble()*

These methods return a *Collector* which collects average of input elements.

Example : Collecting average percentage

```
Double averagePercentage =
studentList.stream().collect(Collectors.averagingDouble(Student::getPercentage));

System.out.println(averagePercentage);

//Output : 81.5
```

3.11) *summarizingInt(), summarizingLong(), summarizingDouble()*

These methods return a special class called *Int/Long/ DoubleSummaryStatistics* which contain statistical information like sum, max, min, average etc of input elements.

Example : Extracting highest, lowest and average of percentage of students

```
DoubleSummaryStatistics studentStats =
studentList.stream().collect(Collectors.summarizingDouble(Student::getPercentage));

System.out.println("Highest Percentage : "+studentStats.getMax());

System.out.println("Lowest Percentage : "+studentStats.getMin());

System.out.println("Average Percentage : "+studentStats.getAverage());

//Output :

//Highest Percentage : 92.8
//Lowest Percentage : 71.5
//Average Percentage : 81.5
```

3.12) *Collectors.groupingBy()* :

This method groups the input elements according supplied classifier and returns the results in a *Map*.

Example : Grouping the students by subject

```
Map<String, List<Student>> studentsGroupedBySubject =
studentList.stream().collect(Collectors.groupingBy(Student::getSubject));

System.out.println(studentsGroupedBySubject);

//Output :

//{Economics=[Paul-11-Economics-78.9, Soumya-15-Economics-77.5],
// Literature=[Xiano-14-Literature-71.5],
// Computer Science=[Zevin-12-Computer Science-91.2, Nihira-17-Computer Science-84.6],
// Mathematics=[Asif-16-Mathematics-89.4, Vijay-19-Mathematics-92.8],
// History=[Harish-13-History-83.7, Mitshu-18-History-73.5, Harry-20-History-71.9]}
```

3.13) *Collectors.partitioningBy()* :

This method partitions the input elements according to supplied *Predicate* and returns a *Map<Boolean, List<T>>*. Under the *true* key, you will find elements which match given *Predicate* and under the *false* key, you will find the elements which doesn't match given *Predicate*.

Example : Partitioning the students who got above 80.0% from who don't.

```
Map<Boolean, List<Student>> studentspartionedByPercentage =
studentList.stream().collect(Collectors.partitioningBy(student ->
student.getPercentage() > 80.0));

System.out.println(studentspartionedByPercentage);

//Output :

// {false=[Paul-11-Economics-78.9, Xiano-14-Literature-71.5, Soumya-15-Economics-77.5,
Mitshu-18-History-73.5, Harry-20-History-71.9],
//  true=[Zevin-12-Computer Science-91.2, Harish-13-History-83.7, Asif-16-Mathematics-
89.4, Nihira-17-Computer Science-84.6, Vijay-19-Mathematics-92.8]}
```

3.14) *Collectors.collectingAndThen()* :

This is a special method which lets you to perform one more action on the result after collecting the result.

Example : Collecting first three students into *List* and making it unmodifiable

```
List<Student> first3Students =
studentList.stream().limit(3).collect(Collectors.collectingAndThen(Collectors.toList(),
Collections::unmodifiableList));

System.out.println(first3Students);

//Output :

//[Paul-11-Economics-78.9, Zevin-12-Computer Science-91.2, Harish-13-History-83.7]
```

Solving Real Time Queries Using Java 8 Features –

Employee Management System

Employee Class:

```
class Employee
{
    int id;
    String name;
    int age;
    String gender;
    String department;
    int yearOfJoining;
    double salary;

    public Employee(int id, String name, int age, String gender, String department,
int yearOfJoining, double salary)
    {
        this.id = id;
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.department = department;
        this.yearOfJoining = yearOfJoining;
        this.salary = salary;
    }

    public int getId()
    {
        return id;
    }

    public String getName()
    {
        return name;
    }

    public int getAge()
    {
        return age;
    }

    public String getGender()
    {
        return gender;
    }

    public String getDepartment()
    {
        return department;
    }

    public int getYearOfJoining()
    {
        return yearOfJoining;
    }

    public double getSalary()
    {
        return salary;
    }

    @Override
```

```
    public String toString()
    {
        return "Id : "+id
                +", Name : "+name
                +", age : "+age
                +", Gender : "+gender
                +", Department : "+department
                +", Year Of Joining : "+yearOfJoining
                +", Salary : "+salary;
    }
}
```

List of Employees: *employeeList*

```
List<Employee> employeeList = new ArrayList<Employee>();

employeeList.add(new Employee(111, "Jiya Brein", 32, "Female", "HR", 2011, 25000.0));
employeeList.add(new Employee(122, "Paul Niksui", 25, "Male", "Sales And Marketing", 2015, 13500.0));
employeeList.add(new Employee(133, "Martin Theron", 29, "Male", "Infrastructure", 2012, 18000.0));
employeeList.add(new Employee(144, "Murali Gowda", 28, "Male", "Product Development", 2014, 32500.0));
employeeList.add(new Employee(155, "Nima Roy", 27, "Female", "HR", 2013, 22700.0));
employeeList.add(new Employee(166, "Iqbal Hussain", 43, "Male", "Security And Transport", 2016, 10500.0));
employeeList.add(new Employee(177, "Manu Sharma", 35, "Male", "Account And Finance", 2010, 27000.0));
employeeList.add(new Employee(188, "Wang Liu", 31, "Male", "Product Development", 2015, 34500.0));
employeeList.add(new Employee(199, "Amelia Zoe", 24, "Female", "Sales And Marketing", 2016, 11500.0));
employeeList.add(new Employee(200, "Jaden Dough", 38, "Male", "Security And Transport", 2015, 11000.5));
employeeList.add(new Employee(211, "Jasna Kaur", 27, "Female", "Infrastructure", 2014, 15700.0));
employeeList.add(new Employee(222, "Nitin Joshi", 25, "Male", "Product Development", 2016, 28200.0));
employeeList.add(new Employee(233, "Jyothi Reddy", 27, "Female", "Account And Finance", 2013, 21300.0));
employeeList.add(new Employee(244, "Nicolus Den", 24, "Male", "Sales And Marketing", 2017, 10700.5));
employeeList.add(new Employee(255, "Ali Baig", 23, "Male", "Infrastructure", 2018, 12700.0));
employeeList.add(new Employee(266, "Sanvi Pandey", 26, "Female", "Product Development", 2015, 28900.0));
employeeList.add(new Employee(277, "Anuj Chettiar", 31, "Male", "Product Development", 2012, 35700.0));
```

Real Time Queries On *employeeList*

1. **How many male and female employees are there in the organization?**

For queries such as above where you need to group the input elements, use the *Collectors.groupingBy()* method. In this query, we use *Collectors.groupingBy()* method which takes two arguments. We pass *Employee::getGender* as first argument which groups the input elements based on *gender* and *Collectors.counting()* as second argument which counts the number of entries in each group.

```
Map<String, Long> noOfMaleAndFemaleEmployees=
employeeList.stream().collect(Collectors.groupingBy(Employee::getGender,
Collectors.counting()));

System.out.println(noOfMaleAndFemaleEmployees);
```

Output :

{Male=11, Female=6}

2. **Print the name of all departments in the organization?**

Use *distinct()* method after calling *map(Employee::getDepartment)* on the stream. It will return unique departments.

```
employeeList.stream()
            .map(Employee::getDepartment)
            .distinct()
            .forEach(System.out::println);
```

Output :

HR
Sales And Marketing
Infrastructure
Product Development
Security And Transport
Account And Finance

3. What is the average age of male and female employees?

Use same method as query 1 but pass *Collectors.averagingInt(Employee::getAge)* as the second argument to *Collectors.groupingBy()*.

```
Map<String, Double> avgAgeOfMaleAndFemaleEmployees=
employeeList.stream().collect(Collectors.groupingBy(Employee::getGender,
Collectors.averagingInt(Employee::getAge)));

System.out.println(avgAgeOfMaleAndFemaleEmployees);
```

Output : {Male=30.181818181818183, Female=27.166666666666668}

4. Get the details of highest paid employee in the organization?

Use *Collectors.maxBy()* method which returns maximum element wrapped in an *Optional* object based on supplied *Comparator*

```
Optional<Employee> highestPaidEmployeeWrapper=
employeeList.stream().collect(Collectors.maxBy(Comparator.comparingDouble(Employee::get
Salary)));

Employee highestPaidEmployee = highestPaidEmployeeWrapper.get();

System.out.println("Details Of Highest Paid Employee : ");
System.out.println("================================");
System.out.println("ID : "+highestPaidEmployee.getId());
System.out.println("Name : "+highestPaidEmployee.getName());
System.out.println("Age : "+highestPaidEmployee.getAge());
System.out.println("Gender : "+highestPaidEmployee.getGender());
System.out.println("Department : "+highestPaidEmployee.getDepartment());
System.out.println("Year Of Joining :
"+highestPaidEmployee.getYearOfJoining());
System.out.println("Salary : "+highestPaidEmployee.getSalary());
```

Output :

Details Of Highest Paid Employee :
==========
ID : 277
Name : Anuj Chettiar
Age : 31
Gender : Male
Department : Product Development
Year Of Joining : 2012
Salary : 35700.0

5. Get the names of all employees who have joined after 2015?

For such queries which require filtering of input elements, use *Stream.filter()* method which filters input elements according to supplied *Predicate*.

```
employeeList.stream()
            .filter(e -> e.getYearOfJoining() > 2015)
            .map(Employee::getName)
            .forEach(System.out::println);
```

Output :

Iqbal Hussain
Amelia Zoe
Nitin Joshi
Nicolus Den
Ali Baig

6. Count the number of employees in each department?

This query is same as query 1 but here we are grouping the elements by *department*.

```
Map<String, Long> employeeCountByDepartment=
employeeList.stream().collect(Collectors.groupingBy(Employee::getDepartment,
Collectors.counting()));

Set<Entry<String, Long>> entrySet = employeeCountByDepartment.entrySet();

for (Entry<String, Long> entry : entrySet)
{
    System.out.println(entry.getKey()+" : "+entry.getValue());
}
```

Output :

Product Development : 5
Security And Transport : 2
Sales And Marketing : 3
Infrastructure : 3
HR : 2
Account And Finance : 2

7. What is the average salary of each department?

Use the same method as in the above query 6, but here pass *Collectors.averagingDouble(Employee::getSalary)* as second argument to *Collectors.groupingBy()* method.

```
Map<String, Double> avgSalaryOfDepartments=
employeeList.stream().collect(Collectors.groupingBy(Employee::getDepartment,
Collectors.averagingDouble(Employee::getSalary)));

Set<Entry<String, Double>> entrySet = avgSalaryOfDepartments.entrySet();

for (Entry<String, Double> entry : entrySet)
{
    System.out.println(entry.getKey()+" : "+entry.getValue());
}
```

Output :

Product Development : 31960.0
Security And Transport : 10750.25
Sales And Marketing : 11900.166666666666
Infrastructure : 15466.666666666666
HR : 23850.0
Account And Finance : 24150.0

8. Get the details of youngest male employee in the product development department?

For this query, use *Stream.filter()* method to filter male employees in product development department and to find youngest among them, use *Stream.min()* method.

```
Optional<Employee> youngestMaleEmployeeInProductDevelopmentWrapper=
employeeList.stream()
            .filter(e -> e.getGender()=="Male" && e.getDepartment()=="Product
Development")
            .min(Comparator.comparingInt(Employee::getAge));

Employee youngestMaleEmployeeInProductDevelopment =
youngestMaleEmployeeInProductDevelopmentWrapper.get();

System.out.println("Details Of Youngest Male Employee In Product
Development");
System.out.println("-------------------------------------------");

System.out.println("ID : "+youngestMaleEmployeeInProductDevelopment.getId());
System.out.println("Name : "+youngestMaleEmployeeInProductDevelopment.getName());
System.out.println("Age : "+youngestMaleEmployeeInProductDevelopment.getAge());
System.out.println("Year Of Joinging :
"+youngestMaleEmployeeInProductDevelopment.getYearOfJoining());
System.out.println("Salary : "+youngestMaleEmployeeInProductDevelopment.getSalary());
```

Output :

Details Of Youngest Male Employee In Product Development :
—————————————-
ID : 222
Name : Nitin Joshi
Age : 25
Year Of Joinging : 2016
Salary : 28200.0

9. Who has the most working experience in the organization?

For this query, sort *employeeList* by *yearOfJoining* in natural order and first employee will have most working experience in the organization. To solve this query, we will be using *sorted()* and *findFirst()* methods of *Stream*.

```
Optional<Employee> seniorMostEmployeeWrapper=
employeeList.stream().sorted(Comparator.comparingInt(Employee::getYearOfJoining)).findF
irst();

Employee seniorMostEmployee = seniorMostEmployeeWrapper.get();

System.out.println("Senior Most Employee Details :");
System.out.println("---------------------------");
System.out.println("ID : "+seniorMostEmployee.getId());
```

```
System.out.println("Name : "+seniorMostEmployee.getName());
System.out.println("Age : "+seniorMostEmployee.getAge());
System.out.println("Gender : "+seniorMostEmployee.getGender());
System.out.println("Age : "+seniorMostEmployee.getDepartment());
System.out.println("Year Of Joinging : "+seniorMostEmployee.getYearOfJoining());
System.out.println("Salary : "+seniorMostEmployee.getSalary());
```

Output :

Senior Most Employee Details :
————————————-
ID : 177
Name : Manu Sharma
Age : 35
Gender : Male
Age : Account And Finance
Year Of Joinging : 2010
Salary : 27000.0

10. How many male and female employees are there in the sales and marketing team?

This query is same as query 1, but here use *filter()* method to filter sales and marketing employees.

```
Map<String, Long> countMaleFemaleEmployeesInSalesMarketing=
employeeList.stream()
            .filter(e -> e.getDepartment()=="Sales And Marketing")
            .collect(Collectors.groupingBy(Employee::getGender,
Collectors.counting()));

System.out.println(countMaleFemaleEmployeesInSalesMarketing)
```

Output :

{Female=1, Male=2}

11. What is the average salary of male and female employees?

This query is same as query 3 where you have found average age of male and female employees. Here, we will be finding average salary of male and female employees.

```
Map<String, Double> avgSalaryOfMaleAndFemaleEmployees=
employeeList.stream().collect(Collectors.groupingBy(Employee::getGender,
Collectors.averagingDouble(Employee::getSalary)));

System.out.println(avgSalaryOfMaleAndFemaleEmployees);
```

Output :

{Male=21300.090909090908, Female=20850.0}

12. List down the names of all employees in each department?

For this query, we will be using *Collectors.groupingBy()* method by passing *Employee::getDepartment* as an argument.

```
Map<String, List<Employee>> employeeListByDepartment=
employeeList.stream().collect(Collectors.groupingBy(Employee::getDepartment));

Set<Entry<String, List<Employee>>> entrySet = employeeListByDepartment.entrySet();

for (Entry<String, List<Employee>> entry : entrySet)
{
    System.out.println("---------------------------------------");
    System.out.println("Employees In "+entry.getKey() + " : ");
    System.out.println("---------------------------------------");
    List<Employee> list = entry.getValue();
    for (Employee e : list)
    {
        System.out.println(e.getName());
    }
}
```

Output :

```
_____________________
Employees In Product Development :
_____________________
Murali Gowda
Wang Liu
Nitin Joshi
Sanvi Pandey
Anuj Chettiar
_____________________
Employees In Security And Transport :
_____________________
Iqbal Hussain
Jaden Dough
_____________________
Employees In Sales And Marketing :
_____________________
Paul Niksui
Amelia Zoe
Nicolus Den
_____________________
Employees In Infrastructure :
_____________________
Martin Theron
Jasna Kaur
Ali Baig
_____________________
Employees In HR :
_____________________
Jiya Brein
Nima Roy
_____________________
Employees In Account And Finance :
_____________________
Manu Sharma
Jyothi Reddy
```

13. What is the average salary and total salary of the whole organization?

For this query, we use *Collectors.summarizingDouble()* on *Employee::getSalary* which will return statistics of the employee salary like max, min, average and total.

```
DoubleSummaryStatistics employeeSalaryStatistics=
employeeList.stream().collect(Collectors.summarizingDouble(Employee::getSalary));

System.out.println("Average Salary = "+employeeSalaryStatistics.getAverage());

System.out.println("Total Salary = "+employeeSalaryStatistics.getSum());
```

Output :

Average Salary = 21141.235294117647
Total Salary = 359401.0

14. **Separate the employees who are younger or equal to 25 years from those employees who are older than 25 years.**

For this query, we will be using *Collectors.partitioningBy()* method which separates input elements based on supplied *Predicate*.

```
Map<Boolean, List<Employee>> partitionEmployeesByAge=
employeeList.stream().collect(Collectors.partitioningBy(e -> e.getAge() > 25));

Set<Entry<Boolean, List<Employee>>> entrySet = partitionEmployeesByAge.entrySet();

for (Entry<Boolean, List<Employee>> entry : entrySet)
{
    System.out.println("----------------------------");

    if (entry.getKey())
    {
        System.out.println("Employees older than 25 years :");
    }
    else
    {
        System.out.println("Employees younger than or equal to 25 years :");
    }

    System.out.println("----------------------------");

    List<Employee> list = entry.getValue();

    for (Employee e : list)
    {
        System.out.println(e.getName());
    }
}
```

Output :

```
----------------
Employees younger than or equal to 25 years :
----------------
Paul Niksui
Amelia Zoe
Nitin Joshi
Nicolus Den
Ali Baig
----------------
Employees older than 25 years :
----------------
Jiya Brein
Martin Theron
Murali Gowda
Nima Roy
Iqbal Hussain
Manu Sharma
Wang Liu
Jaden Dough
Jasna Kaur
Jyothi Reddy
Sanvi Pandey
Anuj Chettiar
```

15. **Who is the oldest employee in the organization? What is his age and which department he belongs to?**

```
Optional<Employee> oldestEmployeeWrapper =
employeeList.stream().max(Comparator.comparingInt(Employee::getAge));

Employee oldestEmployee = oldestEmployeeWrapper.get();

System.out.println("Name : "+oldestEmployee.getName());
System.out.println("Age : "+oldestEmployee.getAge());
System.out.println("Department : "+oldestEmployee.getDepartment());
```

Output :

Name: Iqbal Hussain
Age: 43
Department: Security And Transport

16. Employee -> id, salary

- **a. get employee having salary < Rs 10000**
- **b. get 10% on base salary as bonus**
- **c. get Total bonus.**

```
empList.stream()
       .filter(employee -> employee.getSalary() <10000)
         .maptoInt(employee -> (employee.getSalary()*10)/100)
         .sum();
```

SOLID Principles Java

In Java, **SOLID principles** are an object-oriented approach that are applied to software structure design. It is conceptualized by **Robert C. Martin** (also known as Uncle Bob). These five principles have changed the world of object-oriented programming, and also changed the way of writing software. It also ensures that the software is modular, easy to understand, debug, and refactor. In this section, we will discuss **SOLID principles in Java** with proper example**.**

The word SOLID acronym for:

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Let's explain the principles one by one in detail.

Single Responsibility Principle

The single responsibility principle states that **every Java class must perform a single functionality.** Implementation of multiple functionalities in a single class mashup the code and if any modification is required may affect the whole class. It precise the code and the code can be easily maintained. Let's understand the single responsibility principle through an example.

Suppose, **Student** is a class having three methods namely **printDetails(), calculatePercentage(),** and **addStudent().** Hence, the Student class has three responsibilities to print the details of students, calculate percentages, and database. By using the single responsibility principle, we can separate these functionalities into three separate classes to fulfill the goal of the principle.

Student.java

```
1. public class Student
2. {
3. public void printDetails();
4. {
5. //functionality of the method
6. }
7. pubic void calculatePercentage();
8. {
9. //functionality of the method
10. }
11. public void addStudent();
12. {
13. //functionality of the method
14. }
15. }
```

The above code snippet violates the single responsibility principle. To achieve the goal of the principle, we should implement a separate class that performs a single functionality only.

Student.java

```
1. public class Student
2. {
3. public void addStudent();
4. {
5. //functionality of the method
6. }
7. }
```

PrintStudentDetails.java

```
1. public class PrintStudentDetails
2. {
3. public void printDetails();
4. {
5. //functionality of the method
6. }
7. }
```

Percentage.java

```
1. public class Percentage
2. {
3. public void calculatePercentage();
4. {
5. //functionality of the method
6. }
7. }
```

Hence, we have achieved the goal of the single responsibility principle by separating the functionality into three separate classes.

Open-Closed Principle

The application or module entities the methods, functions, variables, etc. The open-closed principle states that according to new requirements **the module should be open for extension but closed for modification.** The extension allows us to implement new functionality to the module. Let's understand the principle through an example.

Suppose, **VehicleInfo** is a class and it has the method **vehicleNumber()** that returns the vehicle number.

VehicleInfo.java

```
1. public class VehicleInfo
2. {
3. public double vehicleNumber(Vehicle vcl)
4. {
5. if (vcl instanceof Car)
6. {
7. return vcl.getNumber();
8. if (vcl instanceof Bike)
9. {
10. return vcl.getNumber();
11. }
12. }
```

If we want to add another subclass named Truck, simply, we add one more if statement that violates the open-closed principle. The only way to add the subclass and achieve the goal of principle by overriding the **vehicleNumber**() method, as we have shown below.

VehicleInfo.java

```
1. public class VehicleInfo
2. {
3. public double vehicleNumber()
4. {
5. //functionality
6. }
7. }
8. public class Car extends VehicleInfo
9. {
10. public double vehicleNumber()
11. {
12. return this.getValue();
13. }
```

```
14. public class Car extends Truck
15. {
16. public double vehicleNumber()
17. {
18. return this.getValue();
19. }
```

Similarly, we can add more vehicles by making another subclass extending from the vehicle class. the approach would not affect the existing application.

Liskov Substitution Principle

The Liskov Substitution Principle (LSP) was introduced by **Barbara Liskov**. It applies to inheritance in such a way that the **derived classes must be completely substitutable for their base classes**. In other words, if class A is a subtype of class B, then we should be able to replace B with A without interrupting the behavior of the program.

It extends the open-close principle and also focuses on the behavior of a superclass and its subtypes. We should design the classes to preserve the property unless we have a strong reason to do otherwise. Let's understand the principle through an example.

Student.java

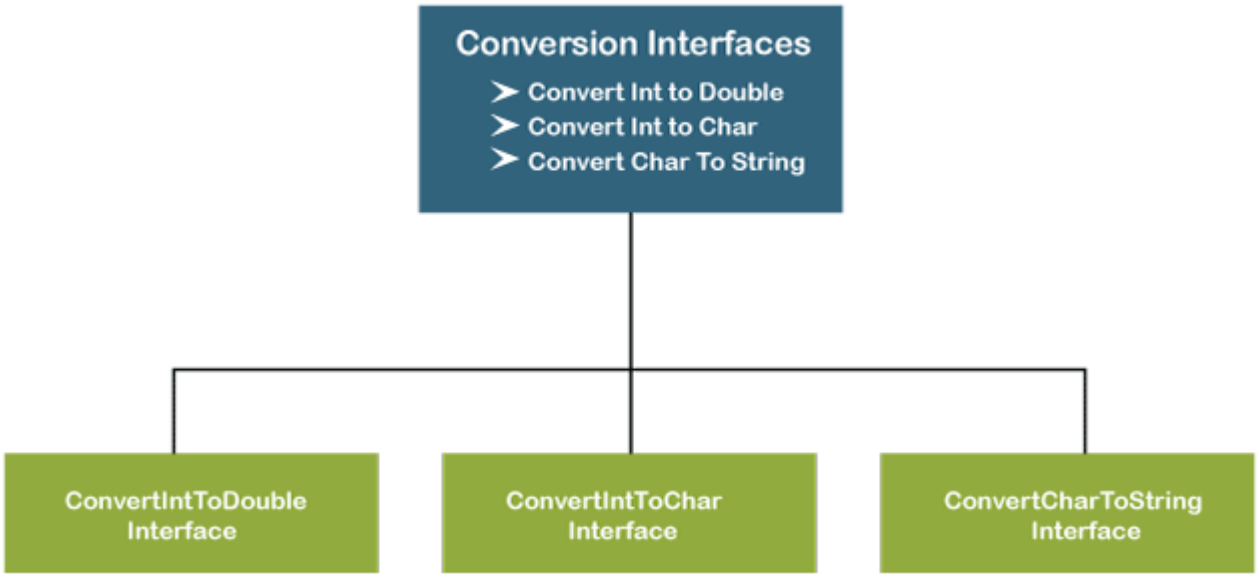
```
1. public class Student
2. {
3. private double height;
4. private double weight;
5. public void setHeight(double h)
6. {
7. height = h;
8. }
9. public void setWeight(double w)
10. {
11. weight= w;
12. }
13. ....
14. }
15. public class StudentBMI extends Student
16. {
17. public void setHeight(double h)
18. {
19. super.setHeight(h);
20. super.setWeight(w);
21. }
22. public void setWeight(double h)
23. {
24. super.setHeight(h);
25. super.setWeight(w);
26. }
27. }
```

The above classes violated the Liskov substitution principle because the StudentBMI class has extra constraints i.e. height and weight that must be the same. Therefore, the Student class (base class) cannot be replaced by StudentBMI class (derived class).
Hence, substituting the class Student with StudentBMI class may result in unexpected behavior

Interface Segregation Principle

The principle states that the larger interfaces split into smaller ones. Because the implementation classes use only the methods that are required. We should not force the client to use the methods that they do not want to use.

The goal of the interface segregation principle is similar to the single responsibility principle. Let's understand the principle through an example.



Suppose, we have created an interface named **Conversion** having three methods **intToDouble(), intToChar(),** and **charToString()**.

```
1. public interface Conversion
2. {
3. public void intToDouble();
4. public void intToChar();
5. public void charToString();
6. }
```

The above interface has three methods. If we want to use only a method intToChar(), we have no choice to implement the single method. To overcome the problem, the principle allows us to split the interface into three separate ones.

```
1. public interface ConvertIntToDouble
2. {
3. public void intToDouble();
4. }
5. public interface ConvertIntToChar
6. {
7. public void intToChar();
8. }
9. public interface ConvertCharToString
10.{
11.public void charToString();
12.}
```

Now we can use only the method that is required. Suppose, we want to convert the integer to double and character to string then, we will use only the methods **intToDouble()** and **charToString().**

```
1. public class DataTypeConversion implements ConvertIntToDouble, ConvertCharToString
```

```
{
public void intToDouble()
{
//conversion logic
}
public void charToString()
{
//conversion logic
}
}
```

Dependency Inversion Principle

The principle states that we must use abstraction (abstract classes and interfaces) instead of concrete implementations. High-level modules should not depend on the low-level module but both should depend on the abstraction. Because the abstraction does not depend on detail but the detail depends on abstraction. It decouples the software. Let's understand the principle through an example.

```
public class WindowsMachine
{
//functionality
}
```

It is worth, if we have not keyboard and mouse to work on Windows. To solve this problem, we create a constructor of the class and add the instances of the keyboard and monitor. After adding the instances, the class looks like the following:

```
public class WindowsMachine
{
public final keyboard;
public final monitor;
public WindowsMachine()
{
monitor = new monitor();  //instance of monitor class
keyboard = new keyboard(); //instance of keyboard class
}
}
```

Now we can work on the Windows machine with the help of a keyboard and mouse. But we still face the problem. Because we have tightly coupled the three classes together by using the new keyword. It is hard o test the class windows machine.

To make the code loosely coupled, we decouple the WindowsMachine from the keyboard by using the Keyboard interface and this keyword.