What is Stream in java :

Streams are a powerful feature introduced in Java 8 that provide a declarative and functional approach to process collections of data. They allow you to perform various operations on a sequence of elements, such as filtering, mapping, sorting, and aggregating, in a concise and efficient manner. Let's explore streams in depth:

1. Introduction to Streams:
    - A Stream in Java is a sequence of elements that can be processed in parallel or sequentially.
    - Streams are not data structures; they are a pipeline through which data is processed.
    - Streams support functional-style operations to process data, allowing for more concise and expressive code.
    - Streams are designed to work with large data sets and provide performance benefits by leveraging parallel processing.
2. Creating Streams: There are several ways to create streams in Java:
    - From a collection using the `stream()` or `parallelStream()` methods:

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
Stream<String> stream = names.stream();
```

- From an array using the `Arrays.stream()` method:

```
int[] numbers = {1, 2, 3, 4, 5};
IntStream stream = Arrays.stream(numbers);
```

- From a stream builder using the `Stream.Builder` class:

```
Stream.Builder<String> builder = Stream.builder();
builder.add("Hello").add("World");
Stream<String> stream = builder.build();
```

3. Intermediate Operations: Intermediate operations are operations that can be applied to a stream to transform or filter the data. Some common intermediate operations include:
    - `filter(predicate)`: Filters the stream based on a given predicate.
    - `map(mapper)`: Transforms each element of the stream using a given mapper function.

    - `flatMap(mapper)`: Transforms each element into a stream of elements and flattens the resulting streams into a single stream.
    - `distinct()`: Removes duplicate elements from the stream.
    - `sorted()`: Sorts the elements of the stream.
4. Terminal Operations: Terminal operations are operations that produce a result or a side effect. They trigger the processing of the stream and produce a final result or a side effect. Some common terminal operations include:
    - `forEach(consumer)`: Performs an action on each element of the stream.
    - `collect(collector)`: Collects the elements of the stream into a collection or other data structure.
    - `reduce(identity, accumulator)`: Performs a reduction on the elements of the stream, using the given identity and accumulator functions.
    - `count()`: Returns the count of elements in the stream.
    - `anyMatch(predicate)`, `allMatch(predicate)`, `noneMatch(predicate)`: Check if any, all, or none of the elements match the given predicate.
    - `findFirst()`, `findAny()`: Returns the first or any element of the stream, respectively.
5. Lazy Evaluation: Streams support lazy evaluation, which means that intermediate operations are not executed until a terminal operation is invoked. This allows for efficient processing by avoiding unnecessary computation.
6. Parallel Processing: Streams can be processed in parallel by invoking the `parallel()` method on the stream. This automatically splits the stream into multiple substreams and processes them concurrently. However, parallel processing should be used with caution, as it may not always result in performance improvements and can introduce synchronization overhead.
7. Chaining Operations: Multiple operations can be chained together to form a pipeline. The result is a new stream that is created by applying the operations in the order they are chained. This allows for the construction of complex data processing pipelines in a concise and readable manner.
8. Stream Example: Let's see an example that demonstrates the use of streams:

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie",
"David", "Eve");
```

```
long count = names.stream()
                .filter(name -> name.length() > 3)
                .sorted()
                .map(String::toUpperCase)
                .count();

System.out.println("Count: " + count);
```

In this example, the stream is created from the `names` list. The stream is filtered to include only names with a length greater than 3, sorted alphabetically, mapped to uppercase, and finally, the count of elements is obtained.

Streams provide a declarative and functional approach to process data in Java. They enable you to write more concise and expressive code while leveraging parallel processing for better performance. By understanding the concepts and operations available in streams, you can take full advantage of this powerful feature in your Java programs.

Examples in class :

```
package steamdemo;

public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }



    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name+getPersonalId(name);
    }

    private String getPersonalId(String name) {
        return "testId";
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

```
package steamdemo;


import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.stream.Collectors;

public class StreamDemo {
    public static void main(String[] args) {
        Person p1 = new Person("Nikit", 1);
        Person p2 = new Person("Akash", 2);
        Person p3 = new Person("KKKK", 3);
        Person p4 = new Person("tttt", 4);
        Person p5 = new Person("yyyy", 5);

        List<Person> personList =  List.of(p1, p2, p3, p4, p5);

//        List<Person> personList = new ArrayList<>();
//        personList.add(p1);
//        personList.add(p2);
//        personList.add(p3);
//        personList.add(p4);
//        personList.add(p5);

        //output : List<Strring> which contains name
        //this map function is different from map of collection
        long startTime = System.currentTimeMillis();
        System.out.println(System.currentTimeMillis());
        List<String> personNames = personList.stream()
                //filter input is Stream<Person> and output is Stream<Person>
                .map(abc -> abc.getName())
                //mapped a person to its name input Stream<Person> output
Stream<String>
                .filter(def -> def.startsWith("A") || def.startsWith("a"))

//                .map(new Function<Person, String>() {
//                            public Object apply(Person abc){
//                                return abc.getName();
//                            }
//                        })
//                .filter(new Predicate<String>() {
//                    @Override
//                    public boolean test(String def) {
//                        return def.startsWith("A") || def.startsWith("a");
//                    }
//                })
                //input Stream<String> and retruned Stream<String>
                .collect(Collectors.toList());
```

```java
        long timeTaken = System.currentTimeMillis() - startTime;
        System.out.println("time taken : "+timeTaken);

        Set<String> personNameSet = personList.parallelStream()
                .map((abc) -> abc.getName())//mapped a person to its name input
Stream<Person> output Stream<String>
                .filter((def) -> def.startsWith("A") || def.startsWith("a"))
                //input Stream<String> and retruned Stream<String>
                .collect(Collectors.toSet());

        Set<String> personNames2 = new HashSet<>();
        for(Person p : personList){
            if(p.getName().startsWith("A")) {
                personNames2.add(p.getName());
            }
        }

        System.out.println("input : "+personList);
        System.out.println("output : "+personNames);

    }
}
```

Assignment question :
Createa stream of person list, get output of person names with all capital letter and filter those names which starts with A and B only.