Presents

Java Streams

The Streams Concept

- Java streams are an abstraction of I/O
  - ✓ A stream is used to process a collection of object
  - ✓ Also an abstraction of the idea of a queue
  - ✓ Very similar to general streaming like Kafka
- A stream is not a data structure or collection
  - ✓ It takes objects from a source without altering the original source
- Each object is processed by a pipeline method
  - ✓ Each pipeline method returns a stream
  - ✓ Pipeline methods chain together
  - ✓ Terminal operations collect the result to be returned

Sample Stream

- Demonstrates some pipeline operations
  - ✓ The stream() method wraps the collection in a stream
  - ✓ The map() method applies a function to each item in the stream, in this case the square function
  - ✓ The collect() method terminates the stream by collecting the items back into a List

```
public class Demo1 {

    public static void main(String[] args) {

        Function<Integer,Integer> square = x -> x * x;
        List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8);
        List<Integer> squares = numbers.stream()
                .map(square)
                .collect(Collectors.toList());
        System.out.println(squares);
    }

}
```

Sample Stream

- Notice the use of our functional method in the stream
  - ✓ However, we can replace this with an anonymous method
  - ✓ This one of the primary uses of functional programming

```
public class Demo1 {

    public static void main(String[] args) {

        //Function<Integer,Integer> square = x -> x * x;
        List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8);
        List<Integer> squares = numbers.stream()
                .map(x -> x * x)
                //.map(square)
                .collect(Collectors.toList());
        System.out.println(squares);
    }
```

Intermediate Methods

- Pipeline methods are intermediate methods that take an input from a stream and outputs a stream
- Some of these are:
  - ✓ map(function) – applies the function to each element of the stream
  - ✓ filter(predicate) – keeps the elements that match the predicate, discards the others
  - ✓ sorted() – sorts the stream
  - ✓ Other pipeline methods are in the java.util.streams library

Terminal Methods

- Terminal methods are methods that take an input from a stream and produce a final result
- Terminal methods mark the end of a stream – each stream can have only one terminal method
- Some of these are:
  - ✓ collect(collection) – returns the result of the intermediate operations as a collection (e.g. list, array etc)
  - ✓ forEach(function) – applies the function to each element of the stream – does not produce an output stream
  - ✓ reduce(function) – uses function to collapse a stream into a single value

LearnQuest

ForEach Example

- A list of integers is
  - ✓ Filtered to remove the odd numbers
  - ✓ Squares each value
  - ✓ Sorts the values
  - ✓ Finally prints each value out

```
List<Integer> number = Arrays.asList(4,5,2,8,1,3,6);

number.stream()
        .filter(x → 0 == x % 2)
        .map(x → x * x)
        .sorted()
        .forEach(y → System.out.println(y));
```

Reduce Example

- A list of integers is
  - ✓ Filtered to remove the odd numbers
  - ✓ Squares each value
  - ✓ Finally adds up the values into a sum

```
public static void main(String[] args) {

    List<Integer> numbers = Arrays.asList(4,5,2,8,1,3,6);
    int total = numbers.stream()
            .filter(x -> 0 == x %2)
            .map(x -> x * x)
            .sorted()
            .reduce(0,(sum,val) -> sum + val);
    System.out.println("Total is " + total);

}
```

Lazy Invocation

- Intermediate operations are not always executed
  - ✓ Only if necessary for the terminal operation
  - ✓ In the example below, there is no terminal operation so none of the intermediate methods are executed

```
public class Demo4 {
    static int counter = 0;
    static void count () { Demo4.counter++;}

    public static void main(String[] args) {
        List<String> names = Arrays.asList("asb1", "aush2", "ydh3");

        names.stream()
        .filter(element -> { Demo4.count();return element.contains("2");});

        System.out.println("       Count = " + Demo4.counter);
    }
```

```
blems  @ Javadoc  Declaration  Console ×
nated> Demo4 [Java Application] C:\tools\java\Java18\bin\javaw.exe (Jul. 9, 2022, 10:38:17 a.m. – 10:38:17 a.m.) [pid: 17016]
    Count = 0
```

Lazy Invocation

- The addition of a terminal method causes the intermediate methods to execute.

```
public class Demo4 {
    static int counter = 0;
    static void count () { Demo4.counter++;}

    public static void main(String[] args) {
        List<String> names = Arrays.asList("asb1", "aush2", "ydh3");

        names.stream()
        .filter(element -> { Demo4.count();return element.contains("2");})
        .forEach(x -> System.out.println("       " + x));
        System.out.println("       Count = " + Demo4.counter);
    }

}
```

blems @ Javadoc Declaration Console ×

ated> Demo4 [Java Application] C:\tools\java\Java18\bin\javaw.exe (Jul. 9, 2022, 10:43:05 a.m. – 10:43:05 a.m.) [pid: 10392]

```
aush2
Count = 3
```

Questions

LearnQuest