Java 8

- Oracle released a new version of Java as Java 8 in March 18, 2014.
- It was a revolutionary release of the Java for software development platform.
- It includes various upgrades to the Java programming, JVM, Tools and libraries.
- Some Features:
  - Functional interfaces,
  - Lambda expressions,
  - Method references,
  - Stream API,
  - Optional classes etc
- From Java 8, interfaces can also have concrete methods i.e methods with body along with abstract methods. Eg: stream() is a default method, no need to implement it every time
- Static methods are introduced to interfaces from Java 8, they can be used to perform basic operations and no need to depend on utility classes for that. Eg: Collection interface and Collections utility class

Functional Programming

- Basically, functional programming is a style of writing computer programs that treat computations as evaluating mathematical functions.
- Functional programming contains the following key concepts:
  - Functions as first class objects - That means that you can create an "instance" of a function/method. Eg: Lambda Expressions
  - Pure functions - A function is a pure function if:
    - The execution of the function has no side effects. (no state change of original input parameteres
    - The return value of the function depends only on the input parameters passed to the function.
  - Higher order functions - A function is a higher order function if at least one of the following conditions are met:
    - The function takes one or more functions as parameters.
    - The function returns another function as result.

- Pure functional programming has a set of rules to follow too:
  - No state
  - No side effects
  - Immutable variables
  - Favour recursion over looping (I guess java streams)

- Use OOP when you need to model complex systems with multiple entities and interactions, and when you need to encapsulate data and behavior into reusable components.

- Use FP when you need to perform pure calculations with simple inputs and outputs, and when you need to avoid side effects or state changes.

Functional Interfaces

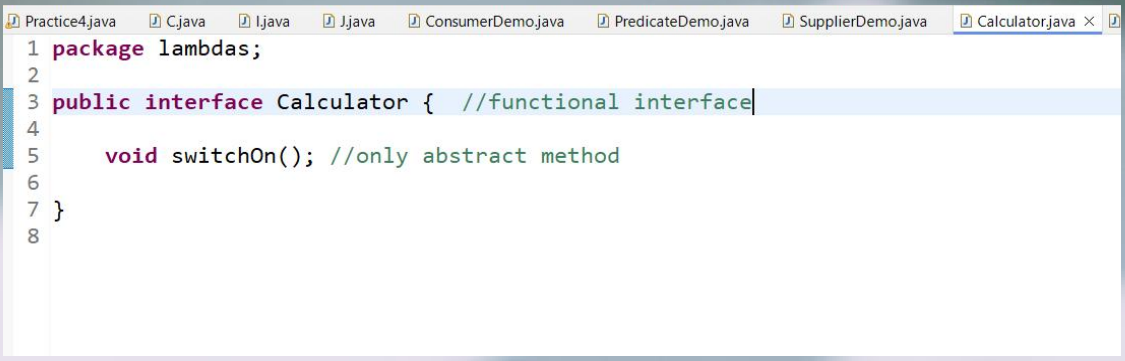
- A functional interface is an interface that contains only one abstract method
- From Java 8 onwards, ***lambda expressions*** can be used to represent the instance of a functional interface
- Examples of Functional interface: Runnable, Comparable, Comparator
- Although they can have any number of default or static methods
- It is additionally recognized as Single Abstract Method Interfaces. In short, they are also known as SAM interfaces
- We can use annotation @FunctionalInterface to ensure that only 1 abstract method is there, but it is not mandatory, it's just a check
- Java 8 included four main kinds of functional interfaces:
  - Consumer (Consumer<T> - void accept(T t))
  - Predicate (Predicate<T> - boolean test(T t))
  - Supplier (Supplier<T> - T get())
  - Function (Function<T, R> - R apply(T t))

  These will be explained in detail later

Lambda Expressions

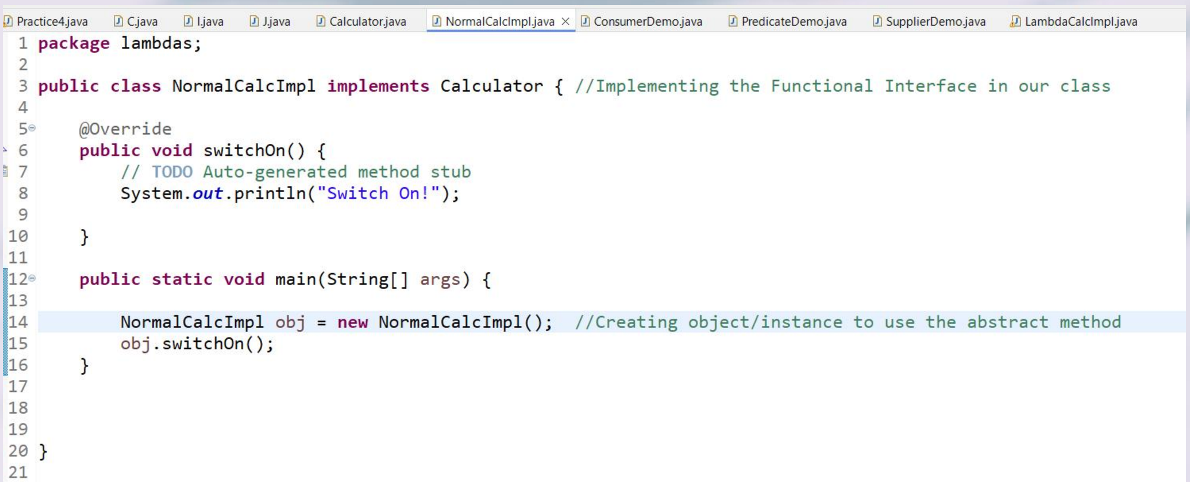
- Lambda expressions basically express instances/object of functional interfaces
- They therefore implement the only abstract method of FI
- Less Coding
- Lambda expressions work only with FIs.
- Lamba function that can be created without belonging to any class
- Lambda syntax: (method parameter) -> {body};
- no need to specify modifier like public or return type like void in it

Eg: Functional Interface



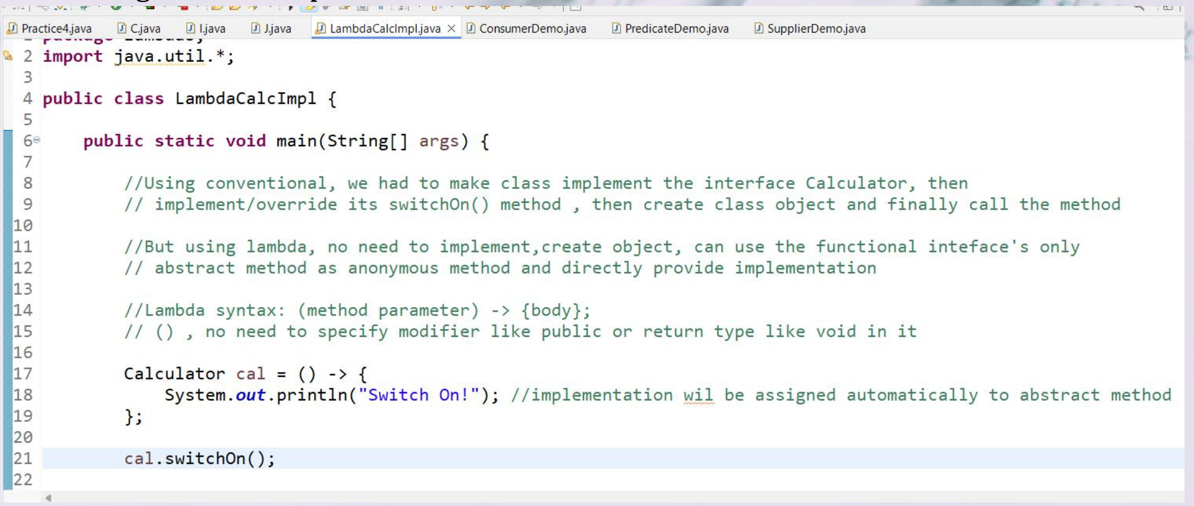
```java
package lambdas;

public interface Calculator {  //functional interface

    void switchOn(); //only abstract method

}

```

Without using Lambda Expressions: (Note- We can also use anonymous class object way too instead of the simple conventional way like below)



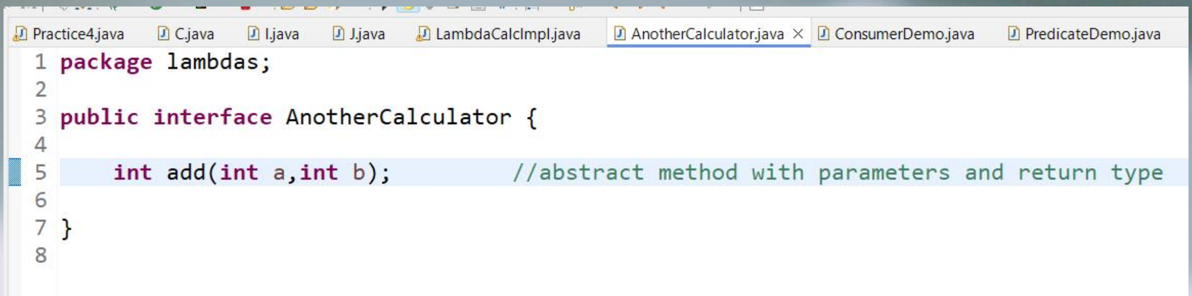
```java
package lambdas;

public class NormalCalcImpl implements Calculator { //Implementing the Functional Interface in our class

    @Override
    public void switchOn() {
        // TODO Auto-generated method stub
        System.out.println("Switch On!");

    }

    public static void main(String[] args) {

        NormalCalcImpl obj = new NormalCalcImpl();  //Creating object/instance to use the abstract method
        obj.switchOn();
    }



}

```

After using Lambda Expressions



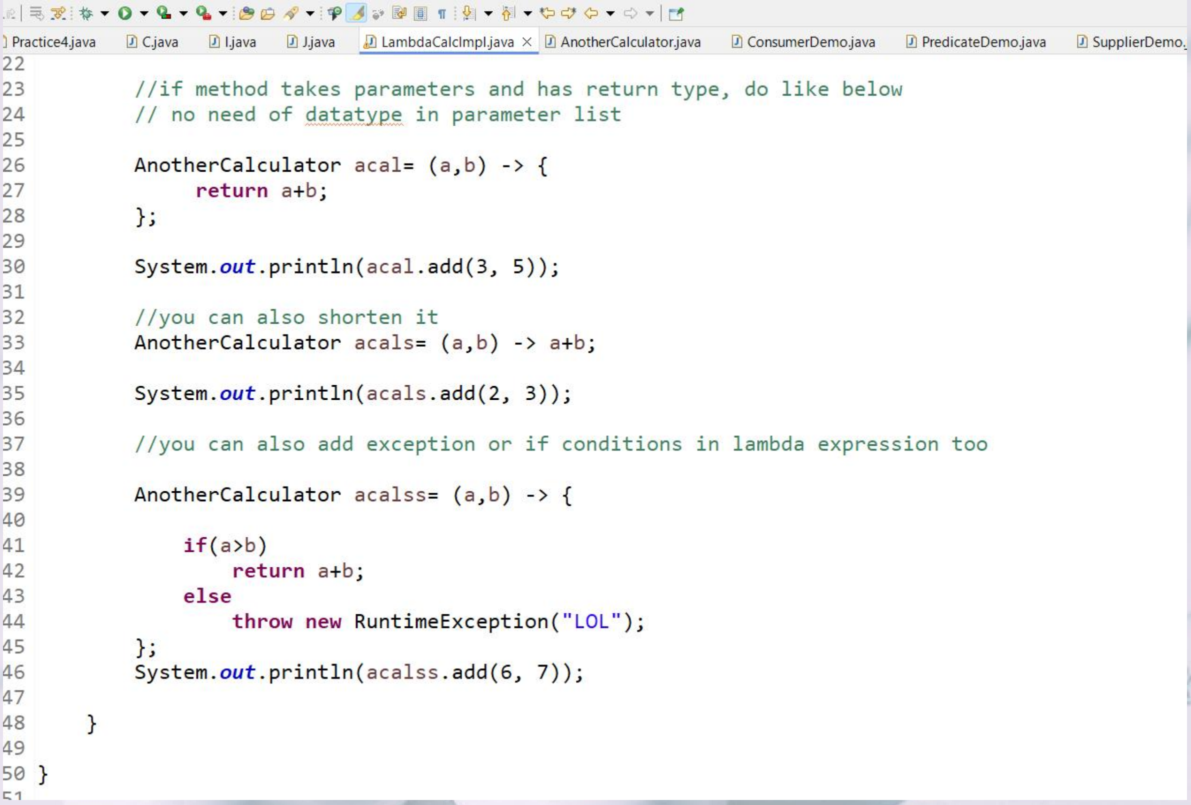
```java
import java.util.*;

public class LambdaCalcImpl {

    public static void main(String[] args) {

        //Using conventional, we had to make class implement the interface Calculator, then
        // implement/override its switchOn() method , then create class object and finally call the method

        //But using lambda, no need to implement,create object, can use the functional inteface's only
        // abstract method as anonymous method and directly provide implementation

        //Lambda syntax: (method parameter) -> {body};
        // () , no need to specify modifier like public or return type like void in it

        Calculator cal = () -> {
            System.out.println("Switch On!"); //implementation wil be assigned automatically to abstract method
        };

        cal.switchOn();

```

Another example if functional interface has an abstract method which takes parameters and has a return type:



```java
package lambdas;

public interface AnotherCalculator {

    int add(int a,int b);          //abstract method with parameters and return type

}

```

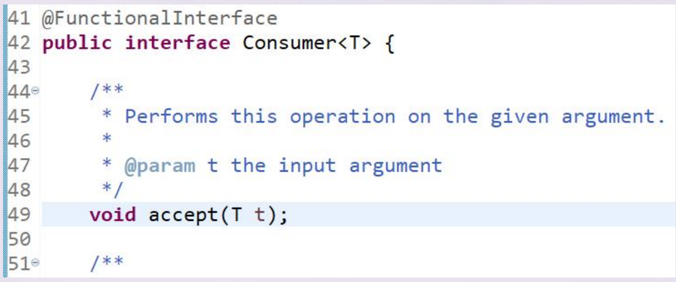
Then, Lambda expressions will be:



```java

        //if method takes parameters and has return type, do like below
        // no need of datatype in parameter list

        AnotherCalculator acal= (a,b) -> {
            return a+b;
        };

        System.out.println(acal.add(3, 5));

        //you can also shorten it
        AnotherCalculator acals= (a,b) -> a+b;

        System.out.println(acals.add(2, 3));

        //you can also add exception or if conditions in lambda expression too

        AnotherCalculator acalss= (a,b) -> {

            if(a>b)
                return a+b;
            else
                throw new RuntimeException("LOL");
        };
        System.out.println(acalss.add(6, 7));

    }

}
```

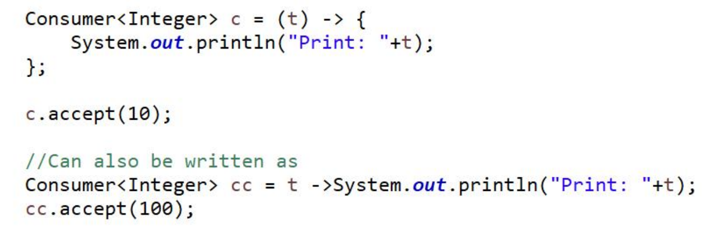
Consumer Functional Interface

- It represents an operation that accepts a single argument and returns no result.
- There are also functional variants of the Consumer — DoubleConsumer, IntConsumer, and LongConsumer. These variants accept primitive values as arguments. “void accept(T t)”
- There is also one more variant of the Consumer interface known as Bi-Consumer. It takes 2 arguments and returns no value



```java
@FunctionalInterface
public interface Consumer<T> {

    /**
     * Performs this operation on the given argument.
     *
     * @param t the input argument
     */
    void accept(T t);

    /**
```

- Eg: Here, ‘accept’ is the abstract method of Consumer FI, ‘t’ is element (int here). We are using Lambda expressions to implement

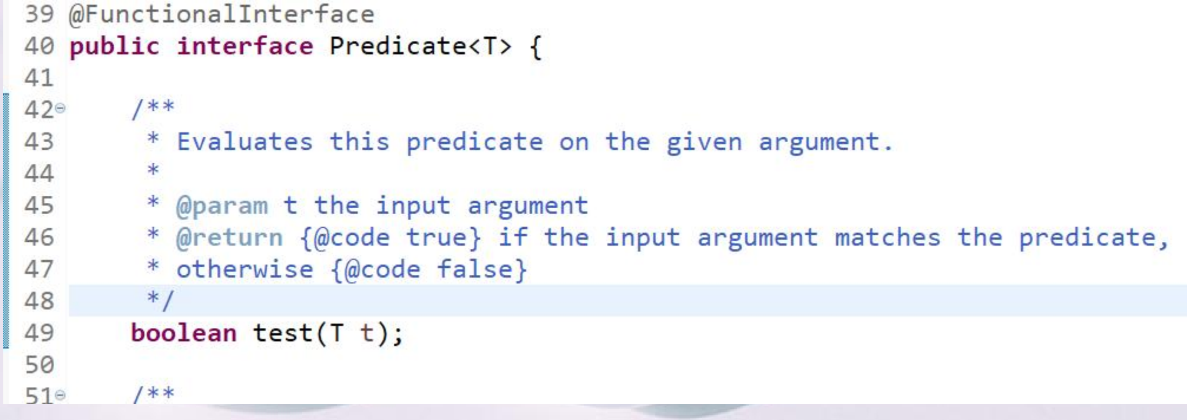


```
Consumer<Integer> c = (t) -> {
    System.out.println("Print: "+t);
};

c.accept(10);

//Can also be written as
Consumer<Integer> cc = t ->System.out.println("Print: "+t);
cc.accept(100);
```

Another example is that stream().forEach() actually uses Consumer interface, so we can use accept() method

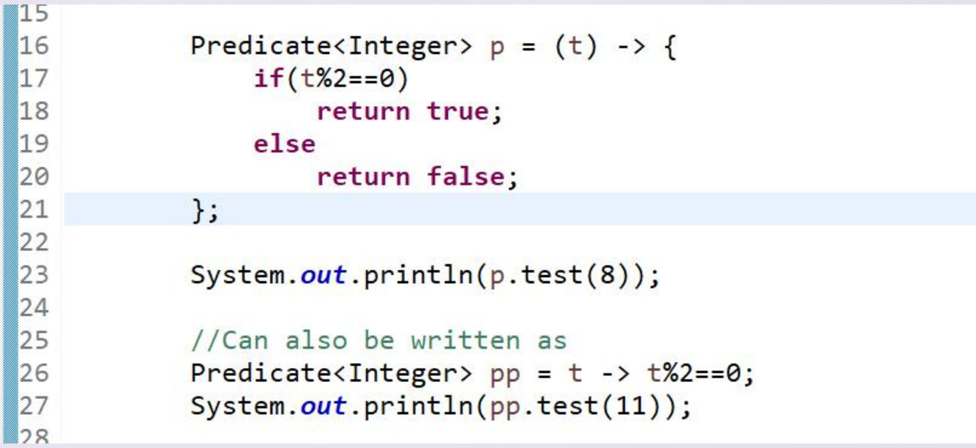
Predicate Functional Interface

- It accepts an argument and returns boolean value as a result. “boolean test(T t)”
- These are IntPredicate, DoublePredicate, and LongPredicate. These types of predicate functional interfaces accept only primitive data types or values as arguments.
- Bi-Predicate is also an extension of the Predicate functional interface, which, instead of one, takes two arguments, does some processing, and returns the boolean value.



```
@FunctionalInterface
public interface Predicate<T> {

    /**
     * Evaluates this predicate on the given argument.
     *
     * @param t the input argument
     * @return {@code true} if the input argument matches the predicate,
     * otherwise {@code false}
     */
    boolean test(T t);

    /**
```

Eg: Here ‘test’ is the abstract method, ‘t’ is element

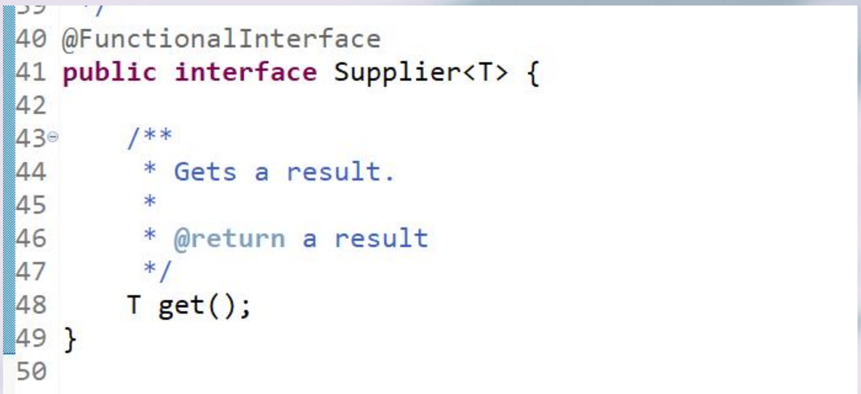


```

        Predicate<Integer> p = (t) -> {
            if(t%2==0)
                return true;
            else
                return false;
        };

        System.out.println(p.test(8));

        //Can also be written as
        Predicate<Integer> pp = t -> t%2==0;
        System.out.println(pp.test(11));
```

Another example is that stream().filter() uses predicate for conditional check

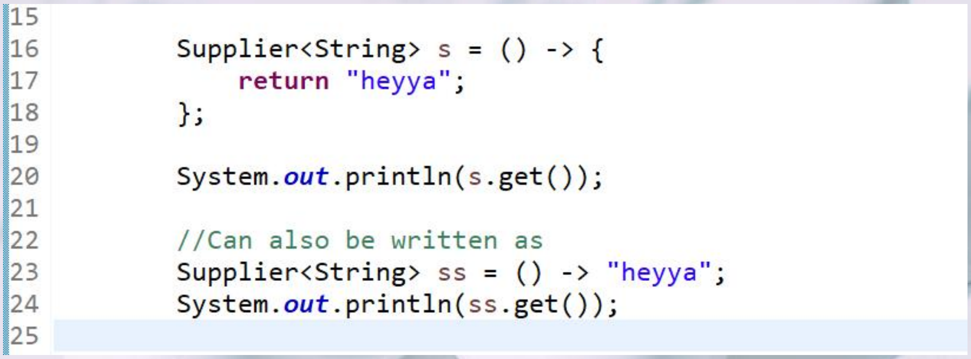
Supplier Functional Interface

- It does not take any input or argument and yet returns a single output
- It represents a supplier of results. “T get()”
- It can be used as a dummy result like if we don’t get any input/parameter, we can print some output
- Best suitable for an operation which creates new objects
- It is generally used in the lazy generation of values
- Supplier functional interfaces are also used for defining the logic for the generation of any sequence



```java
 */
@FunctionalInterface
public interface Supplier<T> {

    /**
     * Gets a result.
     *
     * @return a result
     */
    T get();
}

```

Eg: Here ‘get’ is the abstract method, there is no input argument ‘t’

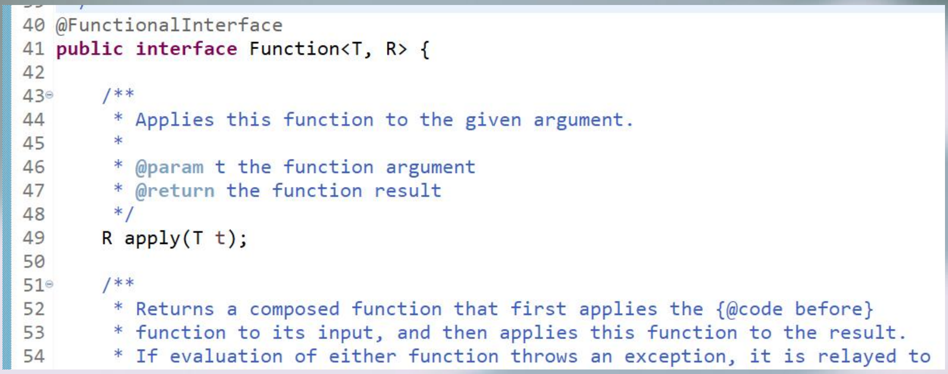


```java

        Supplier<String> s = () -> {
            return "heyya";
        };

        System.out.println(s.get());

        //Can also be written as
        Supplier<String> ss = () -> "heyya";
        System.out.println(ss.get());

```

Also, stream().orElseGet() uses Supplier

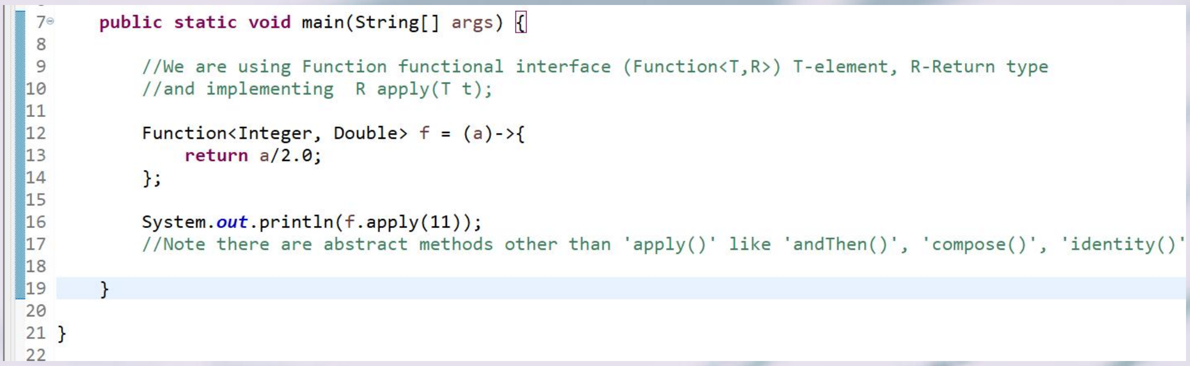
Function Functional Interface

- It represents a function which takes in one argument and produces a result
- Function<T, R> where T: denotes the type of the input argument, R: denotes the return type of the function
- 1 example of abstract method present here is: ‘R apply(T t);’
- Other examples excluding ‘apply()’ are: 'andThen()', 'compose()', 'identity()'



```java
@FunctionalInterface
public interface Function<T, R> {

    /**
     * Applies this function to the given argument.
     *
     * @param t the function argument
     * @return the function result
     */
    R apply(T t);

    /**
     * Returns a composed function that first applies the {@code before}
     * function to its input, and then applies this function to the result.
     * If evaluation of either function throws an exception, it is relayed to
```

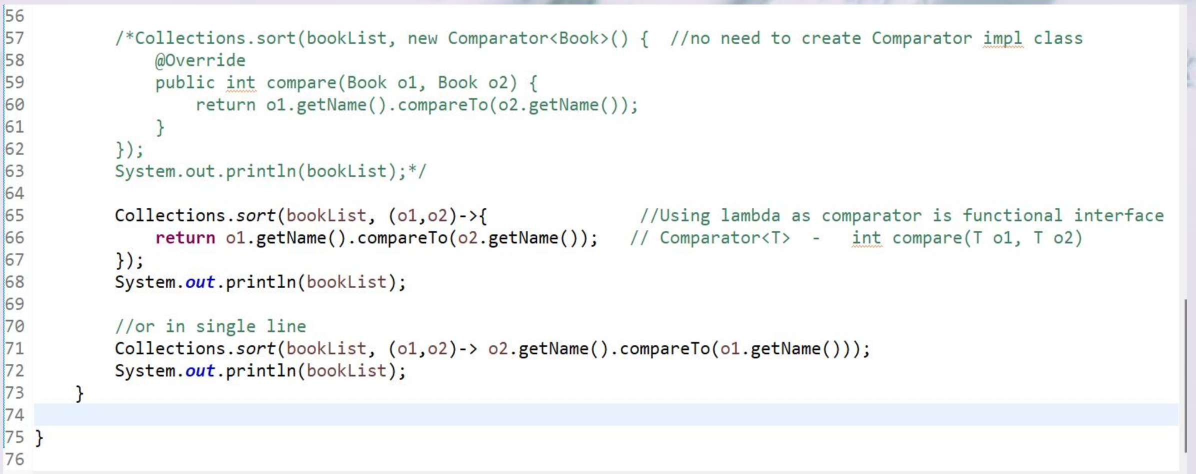
- Example: Here ‘apply’ is the abstract method



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

        //We are using Function functional interface (Function<T,R>) T-element, R-Return type
        //and implementing  R apply(T t);

        Function<Integer, Double> f = (a)->{
            return a/2.0;
        };

        System.out.println(f.apply(11));
        //Note there are abstract methods other than 'apply()' like 'andThen()', 'compose()', 'identity()'

    }

}
```

Lambda expression used in Comparator

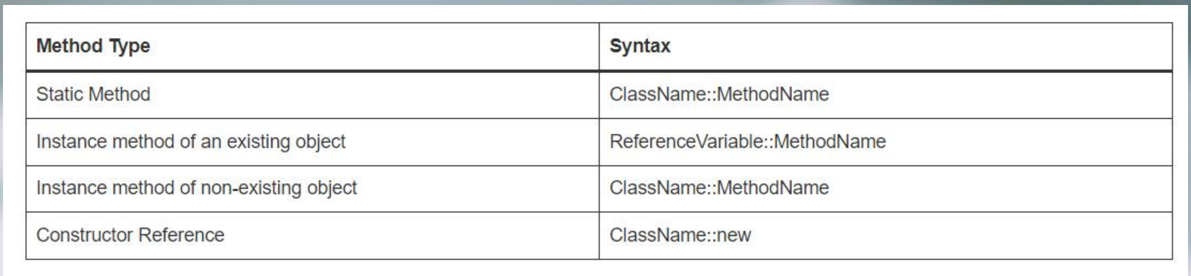
- Here, we have used lambda expression as comparator interface is a functional interface and we have used it to use its ‘compare()’ method



```java

        /*Collections.sort(bookList, new Comparator<Book>() {  //no need to create Comparator impl class
            @Override
            public int compare(Book o1, Book o2) {
                return o1.getName().compareTo(o2.getName());
            }
        });
        System.out.println(bookList);*/

        Collections.sort(bookList, (o1,o2)->{                //Using lambda as comparator is functional interface
            return o1.getName().compareTo(o2.getName());   // Comparator<T>  -   int compare(T o1, T o2)
        });
        System.out.println(bookList);

        //or in single line
        Collections.sort(bookList, (o1,o2)-> o2.getName().compareTo(o1.getName()));
        System.out.println(bookList);
    }

}
```

Method References

- Java 8 method references can be defined as shortened versions of lambda expressions calling a specific method

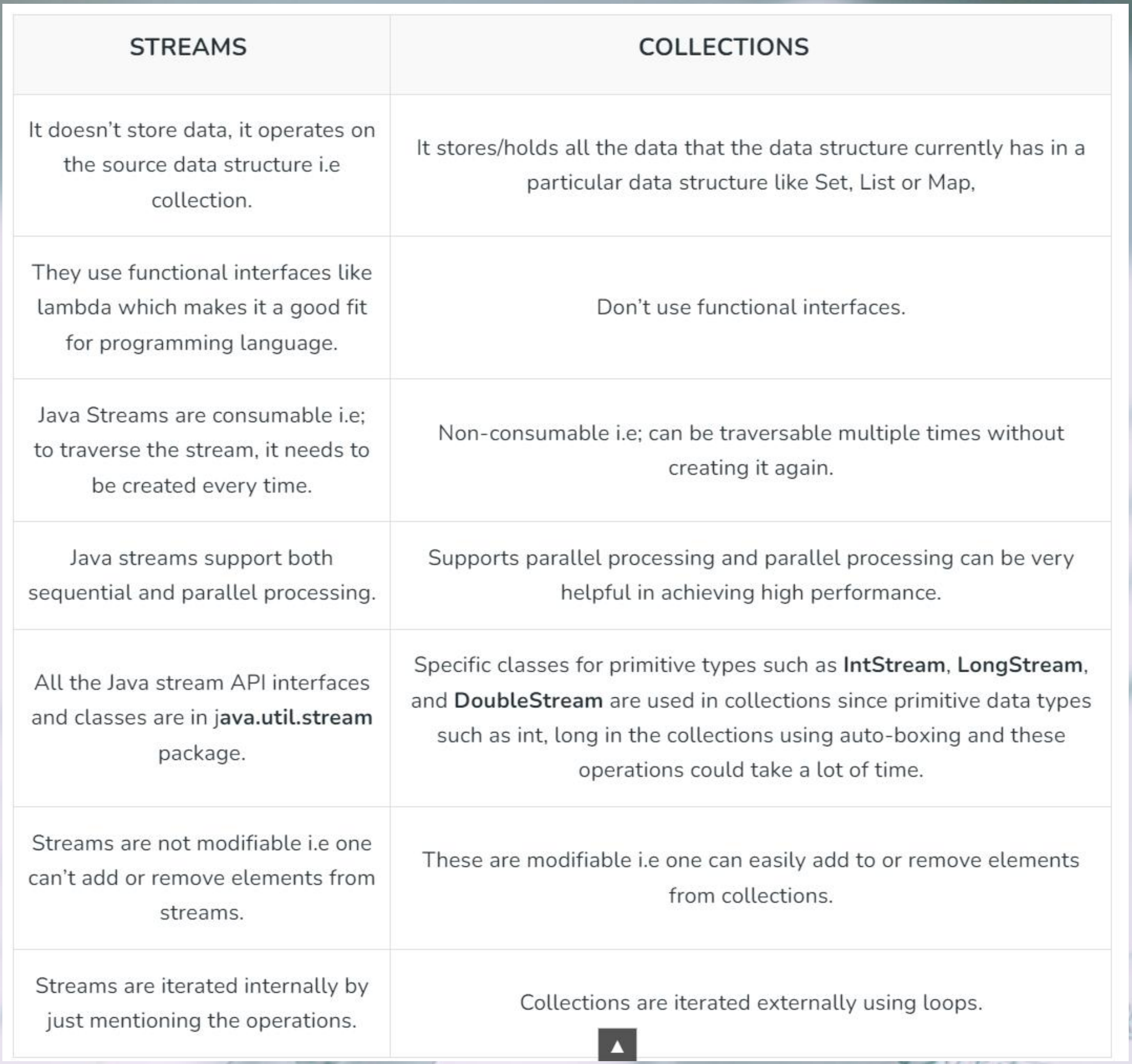
- 

Method Type	Syntax
Static Method	ClassName::MethodName
Instance method of an existing object	ReferenceVariable::MethodName
Instance method of non-existing object	ClassName::MethodName
Constructor Reference	ClassName::new

Streams

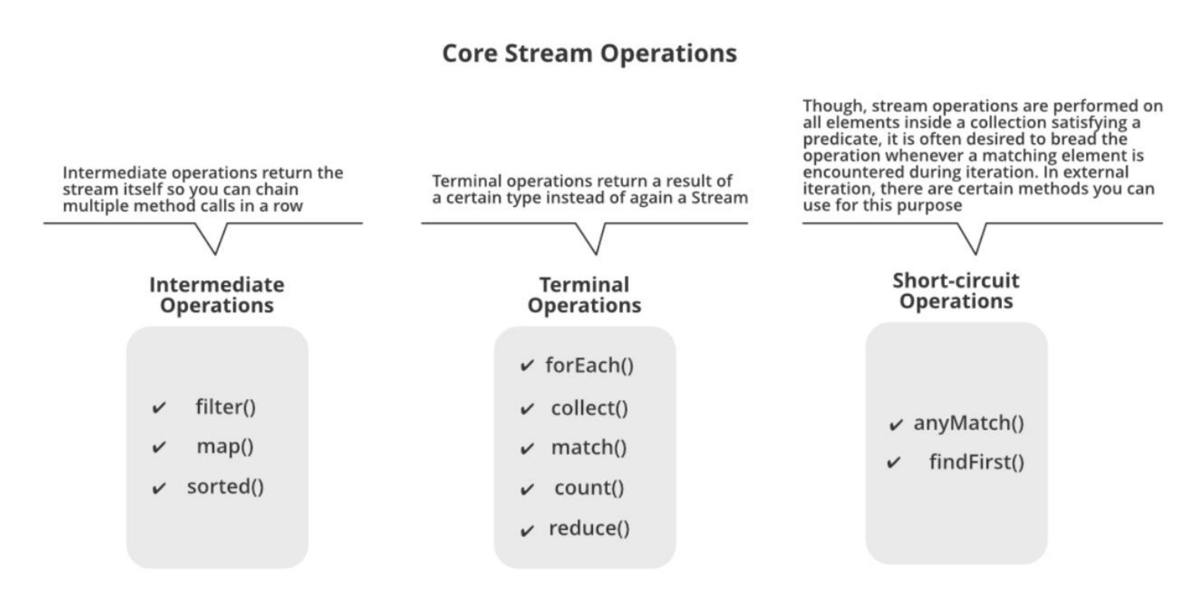
- Stream API is used to process collection of objects
- A stream is a sequence of objects that supports various methods which can be pipelined to produce desired result
- Streams don't change the original data structure, they only provide the result as per the pipelined methods.
- It helps in code reduce, functional programming
- "Stream operations do the iteration implicitly" - 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

- Difference between Streams and Collections



STREAMS	COLLECTIONS
It doesn't store data, it operates on the source data structure i.e collection.	It stores/holds all the data that the data structure currently has in a particular data structure like Set, List or Map,
They use functional interfaces like lambda which makes it a good fit for programming language.	Don't use functional interfaces.
Java Streams are consumable i.e; to traverse the stream, it needs to be created every time.	Non-consumable i.e; can be traversable multiple times without creating it again.
Java streams support both sequential and parallel processing.	Supports parallel processing and parallel processing can be very helpful in achieving high performance.
All the Java stream API interfaces and classes are in **java.util.stream** package.	Specific classes for primitive types such as **IntStream**, **LongStream**, and **DoubleStream** are used in collections since primitive data types such as int, long in the collections using auto-boxing and these operations could take a lot of time.
Streams are not modifiable i.e one can't add or remove elements from streams.	These are modifiable i.e one can easily add to or remove elements from collections.
Streams are iterated internally by just mentioning the operations.	Collections are iterated externally using loops.

- There are mainly 3 types of stream 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



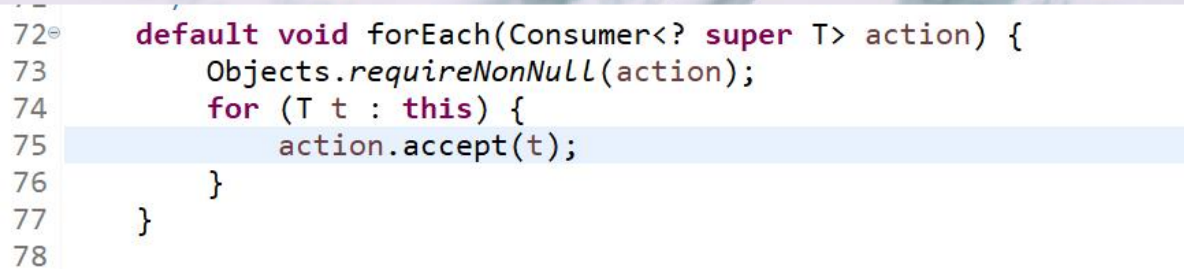
Intermediate operations return the stream itself so you can chain multiple method calls in a row

Terminal operations return a result of a certain type instead of again a Stream

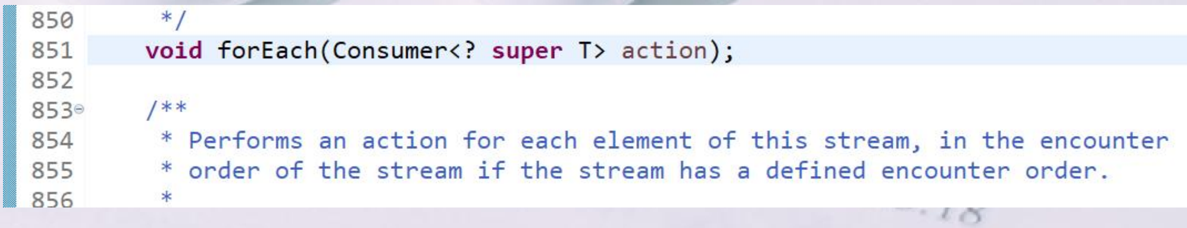
Though, stream operations are performed on all elements inside a collection satisfying a predicate, it is often desired to bread the operation whenever a matching element is encountered during iteration. In external iteration, there are certain methods you can use for this purpose

forEach method

- We use forEach() for iteration
- It is a terminal method
- It uses Consumer interface and it's 'void accept(T t)' method
- Note: forEach() method is available both as Iterator and Stream class (see map iterate below example)
- Below screenshots are from Iterable.class and Stream.class

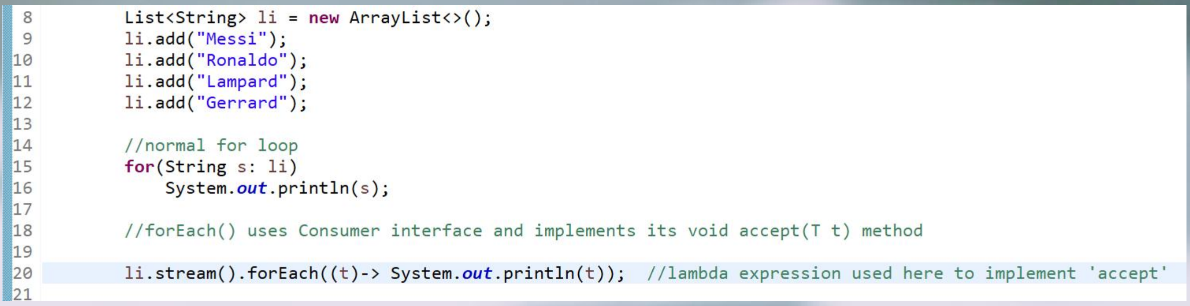


```
    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }

```



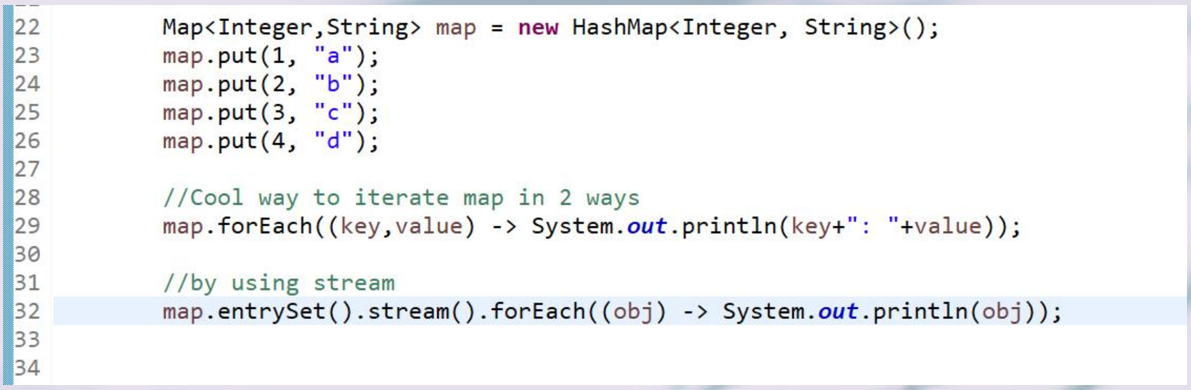
```
     */
    void forEach(Consumer<? super T> action);

    /**
     * Performs an action for each element of this stream, in the encounter
     * order of the stream if the stream has a defined encounter order.
     *
```

- We can use forEach() to easily iterate lists or maps
- Eg: Just write lambda expression inside 'li.stream().forEach()' to implement the 'accept' method from consumer interface



```java
        List<String> li = new ArrayList<>();
        li.add("Messi");
        li.add("Ronaldo");
        li.add("Lampard");
        li.add("Gerrard");

        //normal for loop
        for(String s: li)
            System.out.println(s);

        //forEach() uses Consumer interface and implements its void accept(T t) method

        li.stream().forEach((t)-> System.out.println(t));  //lambda expression used here to implement 'accept'

```

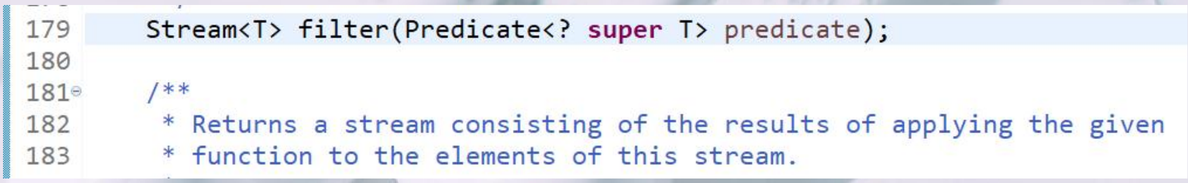
- Similarly in maps



```java
        Map<Integer,String> map = new HashMap<Integer, String>();
        map.put(1, "a");
        map.put(2, "b");
        map.put(3, "c");
        map.put(4, "d");

        //Cool way to iterate map in 2 ways
        map.forEach((key,value) -> System.out.println(key+": "+value));

        //by using stream
        map.entrySet().stream().forEach((obj) -> System.out.println(obj));


```

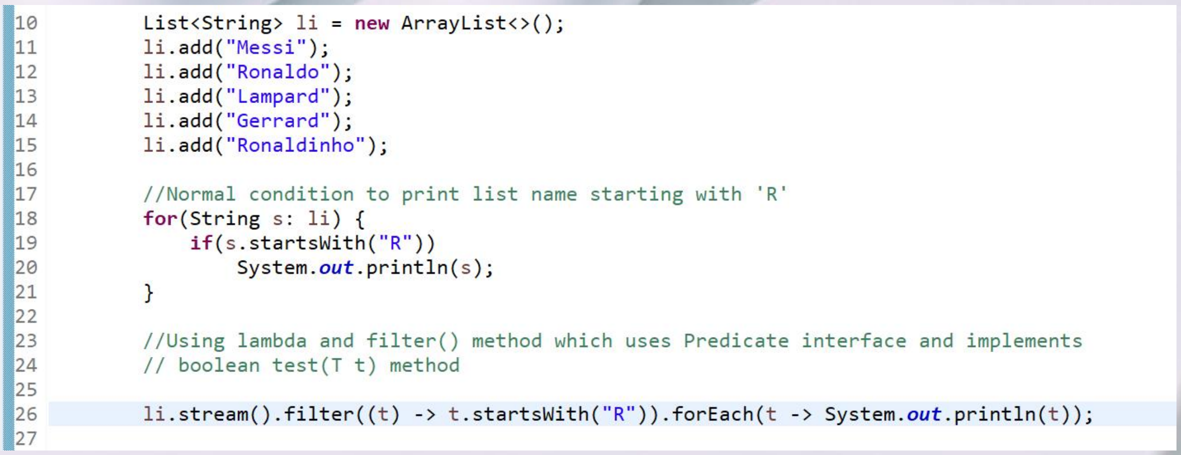
Filter method

- Filter() is used for conditional check
- It is an intermediate method
- It uses Predicate interface and it's 'boolean test(T t)' method



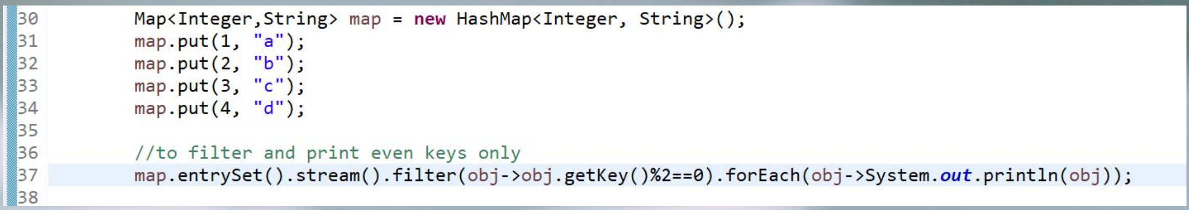
```java
    Stream<T> filter(Predicate<? super T> predicate);

    /**
     * Returns a stream consisting of the results of applying the given
     * function to the elements of this stream.
```

- Example:



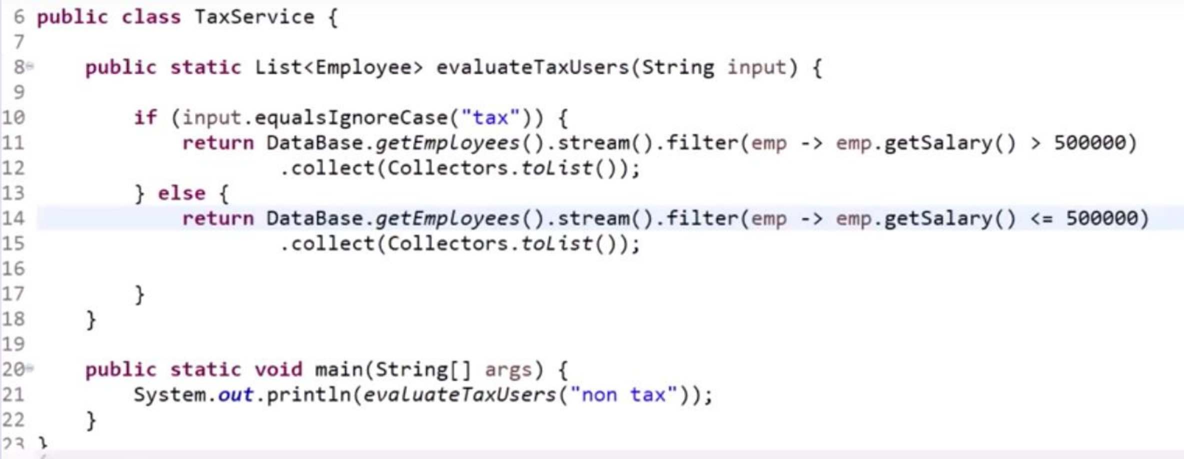
```java
        List<String> li = new ArrayList<>();
        li.add("Messi");
        li.add("Ronaldo");
        li.add("Lampard");
        li.add("Gerrard");
        li.add("Ronaldinho");

        //Normal condition to print list name starting with 'R'
        for(String s: li) {
            if(s.startsWith("R"))
                System.out.println(s);
        }

        //Using lambda and filter() method which uses Predicate interface and implements
        // boolean test(T t) method

        li.stream().filter((t) -> t.startsWith("R")).forEach(t -> System.out.println(t));

```

- Similarly for maps



```
        Map<Integer,String> map = new HashMap<Integer, String>();
        map.put(1, "a");
        map.put(2, "b");
        map.put(3, "c");
        map.put(4, "d");

        //to filter and print even keys only
        map.entrySet().stream().filter(obj->obj.getKey()%2==0).forEach(obj->System.out.println(obj));
```

- Another example using class objects, use .collect(Collectors.toList()) to convert result from stream api to List

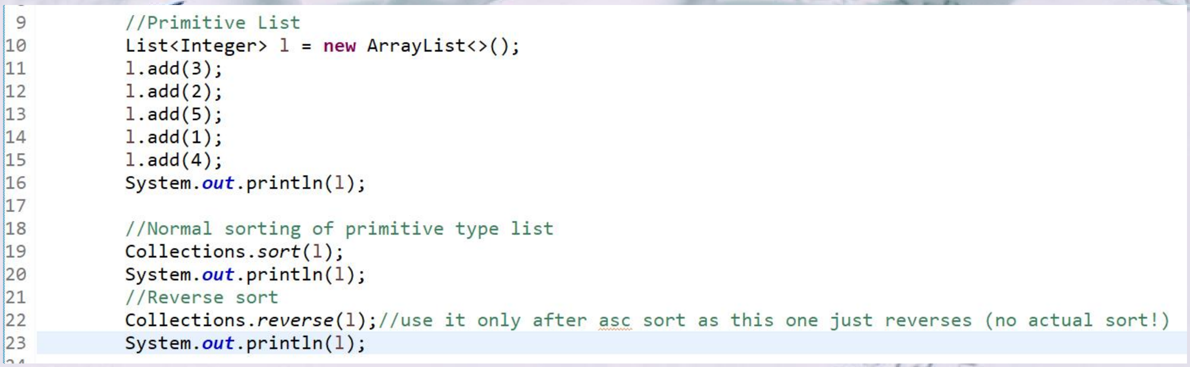


```
public class TaxService {

    public static List<Employee> evaluateTaxUsers(String input) {

        if (input.equalsIgnoreCase("tax")) {
            return DataBase.getEmployees().stream().filter(emp -> emp.getSalary() > 500000)
                    .collect(Collectors.toList());
        } else {
            return DataBase.getEmployees().stream().filter(emp -> emp.getSalary() <= 500000)
                    .collect(Collectors.toList());


        }
    }

    public static void main(String[] args) {
        System.out.println(evaluateTaxUsers("non tax"));
    }
}
```

Sorted() and Comparator.comparing

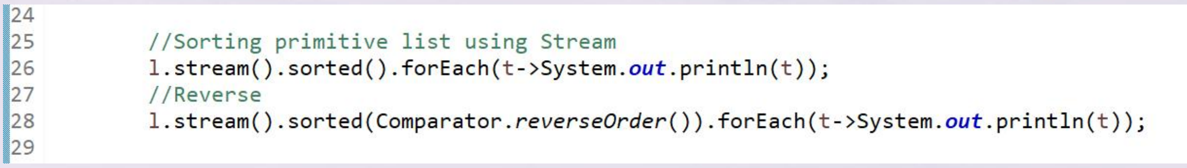
For Lists

- For normal primitive lists, you can use Collections.sort() method for ascending order sort and then Collections.reverse() to reverse the above to get the list in descending order



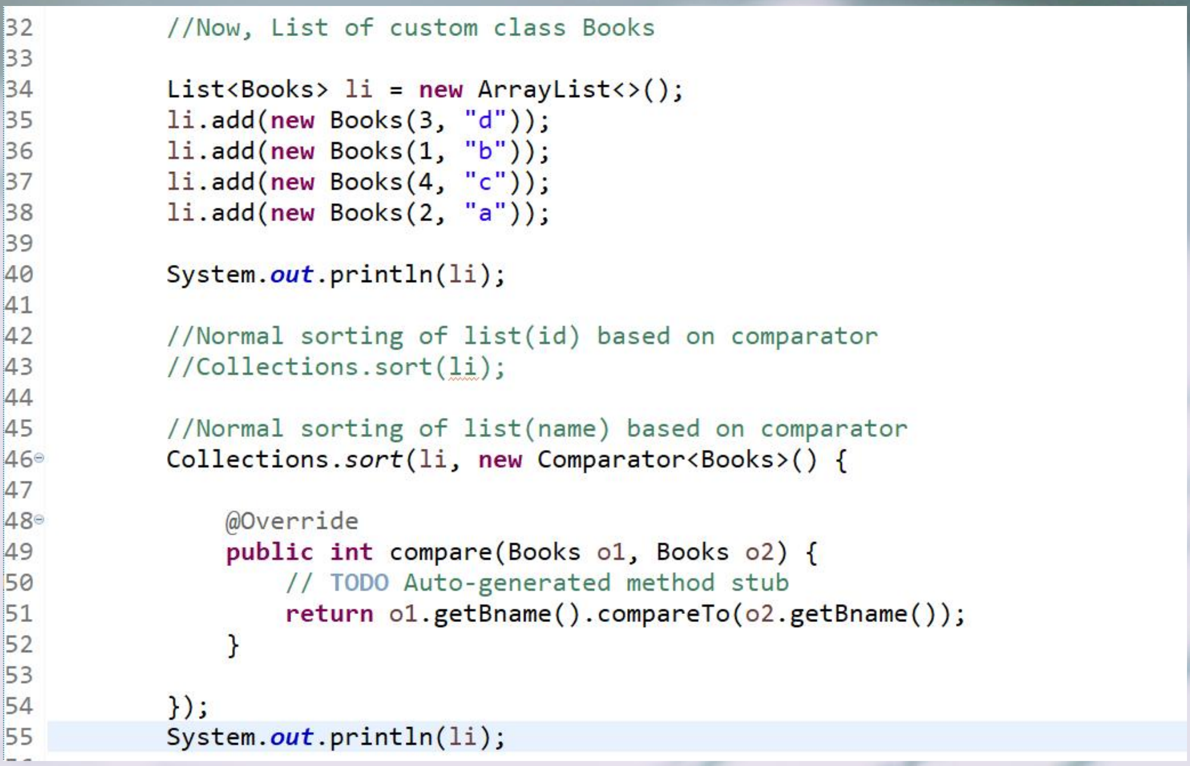
```
        //Primitive List
        List<Integer> l = new ArrayList<>();
        l.add(3);
        l.add(2);
        l.add(5);
        l.add(1);
        l.add(4);
        System.out.println(l);

        //Normal sorting of primitive type list
        Collections.sort(l);
        System.out.println(l);
        //Reverse sort
        Collections.reverse(l);//use it only after asc sort as this one just reverses (no actual sort!)
        System.out.println(l);
```

- Now using streams on same thing above (stream.sorted())



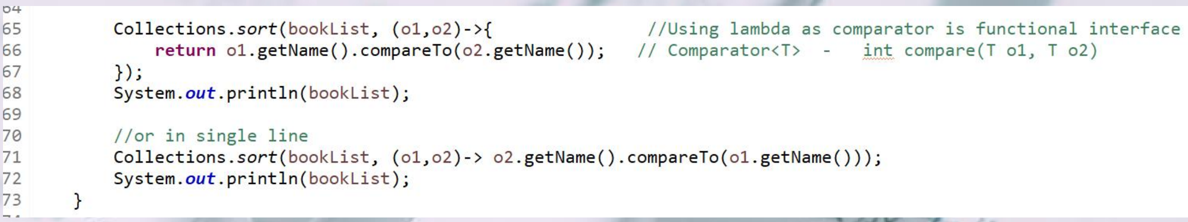
```

        //Sorting primitive list using Stream
        l.stream().sorted().forEach(t->System.out.println(t));
        //Reverse
        l.stream().sorted(Comparator.reverseOrder()).forEach(t->System.out.println(t));

```

- In case of list of objects (Comparing Books)
  - Traditional approach of using Comparator by creating its new object and implementing it's 'compare()' method in Collections.sort()



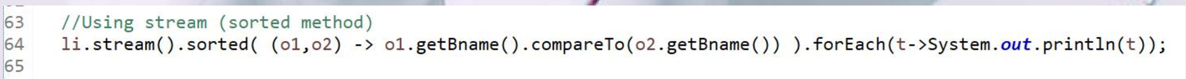
```java
        //Now, List of custom class Books

        List<Books> li = new ArrayList<>();
        li.add(new Books(3, "d"));
        li.add(new Books(1, "b"));
        li.add(new Books(4, "c"));
        li.add(new Books(2, "a"));

        System.out.println(li);

        //Normal sorting of list(id) based on comparator
        //Collections.sort(li);

        //Normal sorting of list(name) based on comparator
        Collections.sort(li, new Comparator<Books>() {

            @Override
            public int compare(Books o1, Books o2) {
                // TODO Auto-generated method stub
                return o1.getBname().compareTo(o2.getBname());
            }

        });
        System.out.println(li);
```

- Same thing above using lambda expressions to make the code shorter (I guess this is still the best)



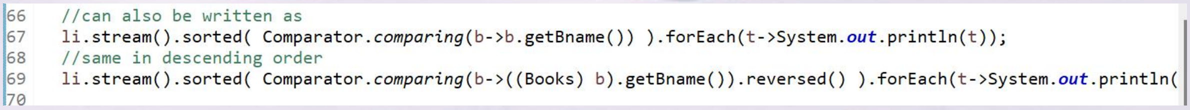
```java
        Collections.sort(bookList, (o1,o2)->{                //Using lambda as comparator is functional interface
            return o1.getName().compareTo(o2.getName());   // Comparator<T>  -   int compare(T o1, T o2)
        });
        System.out.println(bookList);

        //or in single line
        Collections.sort(bookList, (o1,o2)-> o2.getName().compareTo(o1.getName()));
        System.out.println(bookList);
    }
```

- Now using stream().sorted() method (I like this one)



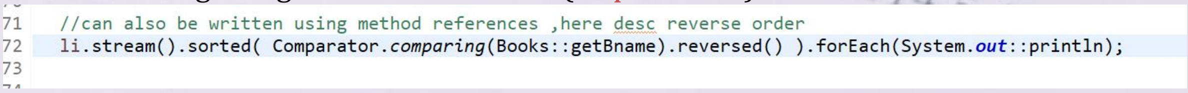
```java
    //Using stream (sorted method)
    li.stream().sorted( (o1,o2) -> o1.getBname().compareTo(o2.getBname()) ).forEach(t->System.out.println(t));
```

- Same thing using stream().sorted(Comparator.comparing)



```java
    //can also be written as
    li.stream().sorted( Comparator.comparing(b->b.getBname()) ).forEach(t->System.out.println(t));
    //same in descending order
    li.stream().sorted( Comparator.comparing(b->((Books) b).getBname()).reversed() ).forEach(t->System.out.println(
```

- Same thing using method references (skip for now)

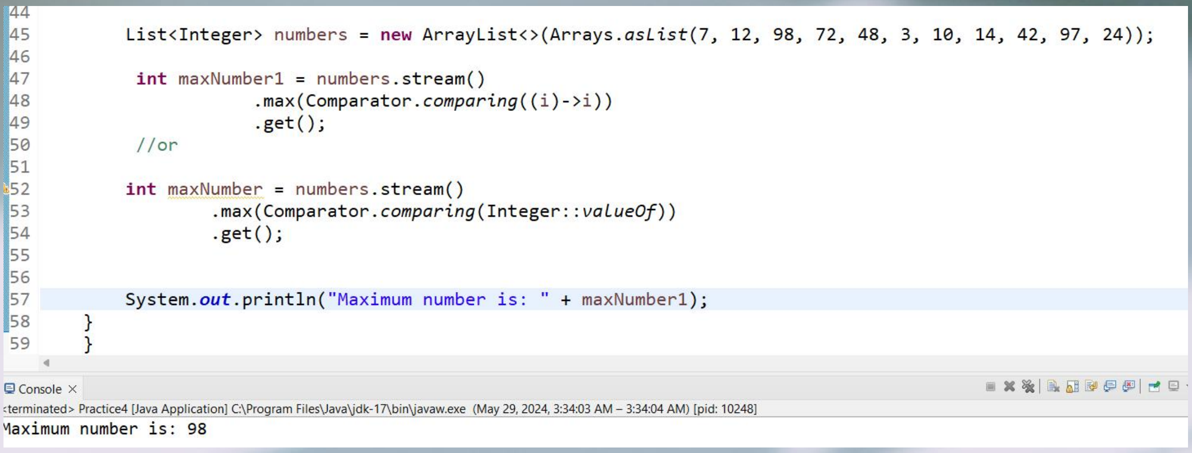


```java
    //can also be written using method references ,here desc reverse order
    li.stream().sorted( Comparator.comparing(Books::getBname).reversed() ).forEach(System.out::println);
```

Note:

Comparator.comparing is just a method that just compares and returns the result, so we just need to add the element/object which is to compared, no need to actually add the comparing logic I think.

Eg:

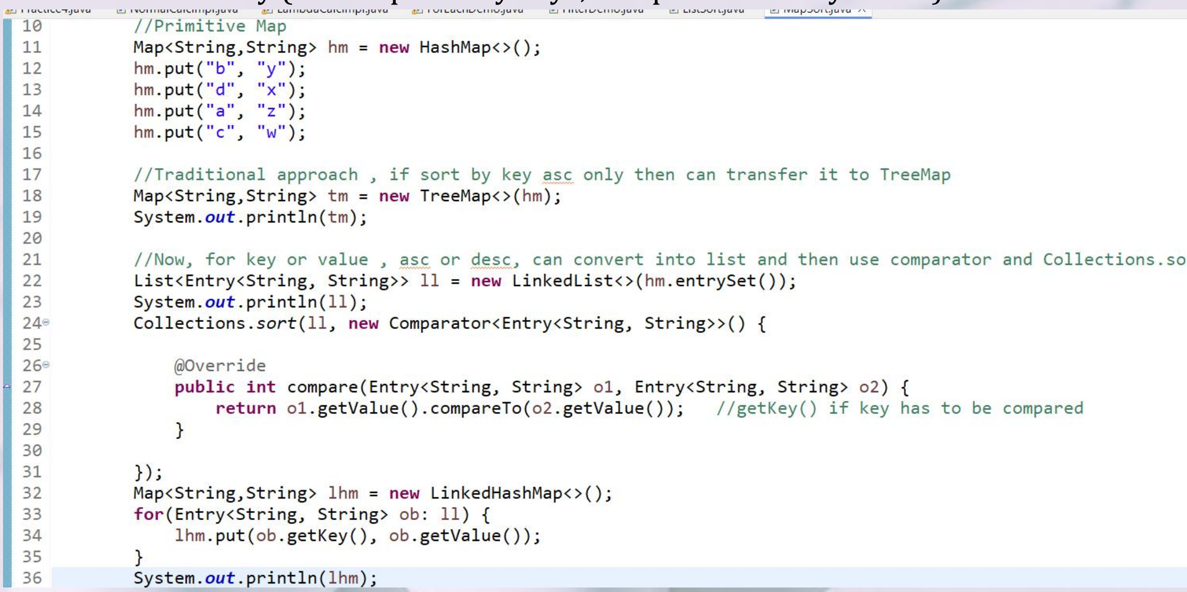


```
        List<Integer> numbers = new ArrayList<>(Arrays.asList(7, 12, 98, 72, 48, 3, 10, 14, 42, 97, 24));

         int maxNumber1 = numbers.stream()
                    .max(Comparator.comparing((i)->i))
                    .get();
         //or

        int maxNumber = numbers.stream()
                .max(Comparator.comparing(Integer::valueOf))
                .get();


        System.out.println("Maximum number is: " + maxNumber1);
    }
    }
```

```
<terminated> Practice4 [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe  (May 29, 2024, 3:34:03 AM – 3:34:04 AM) [pid: 10248]
Maximum number is: 98
```

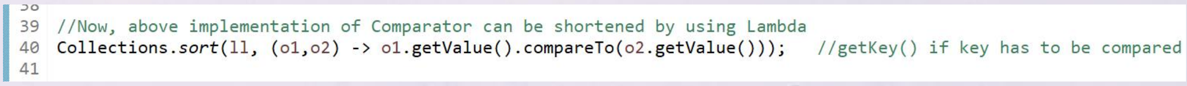
For Maps (Unlike lists, instead of using lambda alone, better to use stream)

- For Primitive maps
  - Traditional way (treemaps   - ‘by key’ , comparator   - ‘by value’)



```
        //Primitive Map
        Map<String,String> hm = new HashMap<>();
        hm.put("b", "y");
        hm.put("d", "x");
        hm.put("a", "z");
        hm.put("c", "w");

        //Traditional approach , if sort by key asc only then can transfer it to TreeMap
        Map<String,String> tm = new TreeMap<>(hm);
        System.out.println(tm);

        //Now, for key or value , asc or desc, can convert into list and then use comparator and Collections.so
        List<Entry<String, String>> ll = new LinkedList<>(hm.entrySet());
        System.out.println(ll);
        Collections.sort(ll, new Comparator<Entry<String, String>>() {

            @Override
            public int compare(Entry<String, String> o1, Entry<String, String> o2) {
                return o1.getValue().compareTo(o2.getValue());   //getKey() if key has to be compared
            }

        });
        Map<String,String> lhm = new LinkedHashMap<>();
        for(Entry<String, String> ob: ll) {
            lhm.put(ob.getKey(), ob.getValue());
        }
        System.out.println(lhm);
```

  - Using Lambda Expression (implementing ‘compare’ method of Comparator FI for both ‘by key’ and ‘by value’ (Note: still converting to Linked List and using Collections.sort(), so not preferred in case of maps)



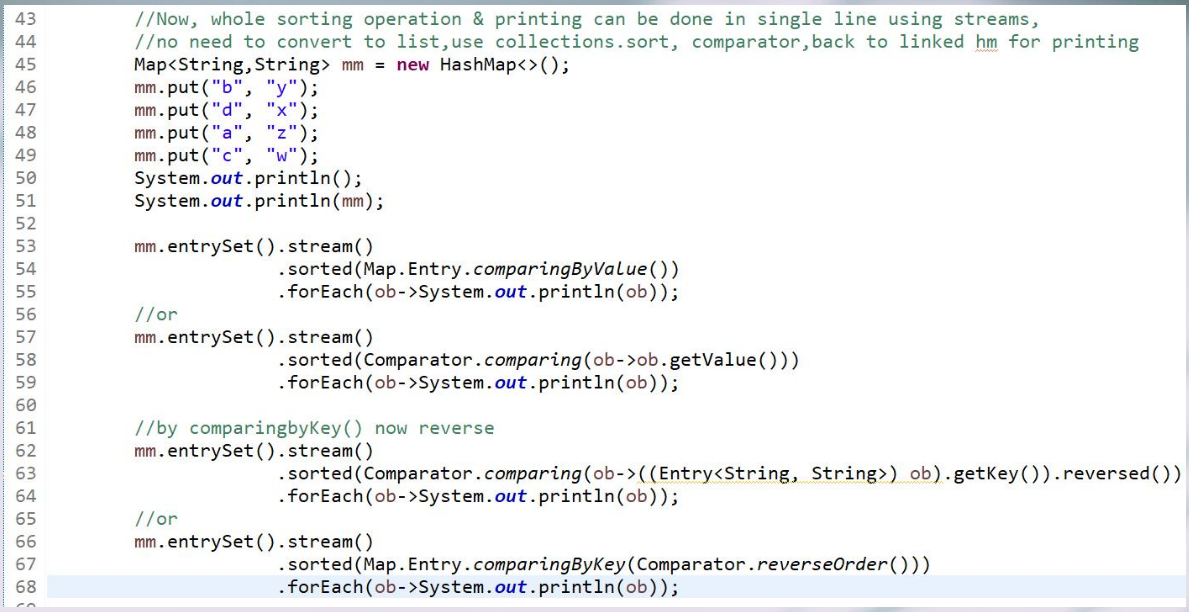
```
//Now, above implementation of Comparator can be shortened by using Lambda
Collections.sort(ll, (o1,o2) -> o1.getValue().compareTo(o2.getValue()));   //getKey() if key has to be compared
```

  - Now, Using Streams, java.stream (prefer this one)

    Many variations are there, anyway no need to convert to list and use Collections.sort(), we can directly use mapName.entrySet().stream() and then

    sorted(Map.entry.comparingByValue(Comparator.reverseOrder())) (better)
    Or
    sorted(Comparator.comparing(ob->ob.getValue()))

    Use the first one as for second one for reverse, u have to use .reversed() and then u need to add cast, will look complicated



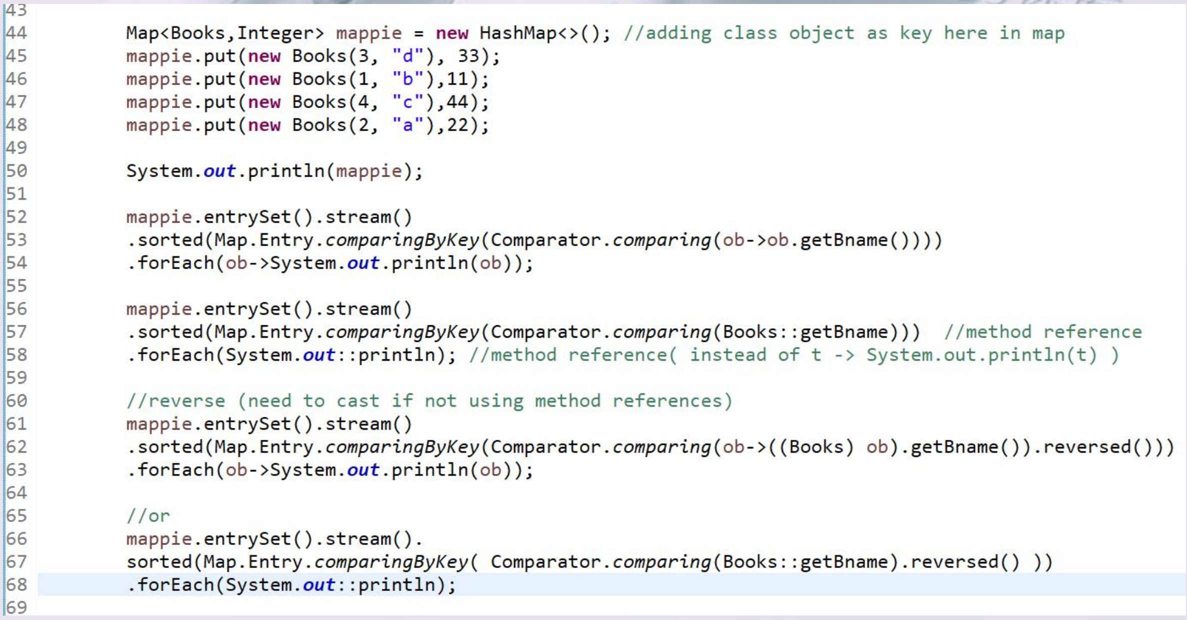
```
        //Now, whole sorting operation & printing can be done in single line using streams,
        //no need to convert to list,use collections.sort, comparator,back to linked hm for printing
        Map<String,String> mm = new HashMap<>();
        mm.put("b", "y");
        mm.put("d", "x");
        mm.put("a", "z");
        mm.put("c", "w");
        System.out.println();
        System.out.println(mm);

        mm.entrySet().stream()
                    .sorted(Map.Entry.comparingByValue())
                    .forEach(ob->System.out.println(ob));
        //or
        mm.entrySet().stream()
                    .sorted(Comparator.comparing(ob->ob.getValue()))
                    .forEach(ob->System.out.println(ob));

        //by comparingbyKey() now reverse
        mm.entrySet().stream()
                    .sorted(Comparator.comparing(ob->((Entry<String, String>) ob).getKey()).reversed())
                    .forEach(ob->System.out.println(ob));
        //or
        mm.entrySet().stream()
                    .sorted(Map.Entry.comparingByKey(Comparator.reverseOrder()))
                    .forEach(ob->System.out.println(ob));
```

- For Custom class object maps

Same like above, only difference is if that map's key or value has multiple class fields, then we just need to specify which field to sort

mapName.entrySet().stream()
.sorted(Map.Entry.comparingByKey) [if only 1 class field]

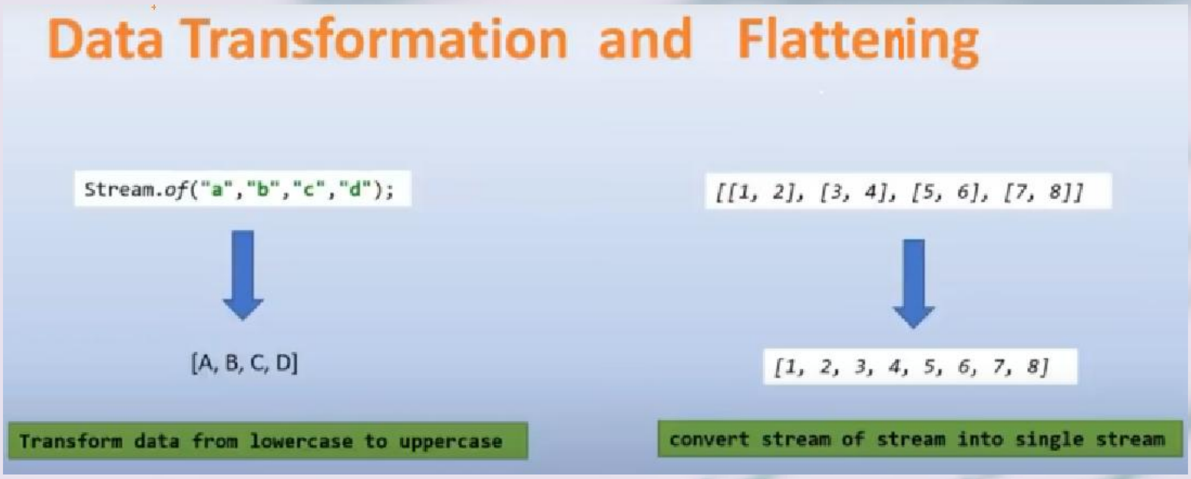
.sorted(Map.Entry.comparingByKey(Comparator.comparing(ob->ob.getBname())) [if multiple class fields, where getBname() is the getter method of class]



```
        Map<Books,Integer> mappie = new HashMap<>(); //adding class object as key here in map
        mappie.put(new Books(3, "d"), 33);
        mappie.put(new Books(1, "b"),11);
        mappie.put(new Books(4, "c"),44);
        mappie.put(new Books(2, "a"),22);

        System.out.println(mappie);

        mappie.entrySet().stream()
        .sorted(Map.Entry.comparingByKey(Comparator.comparing(ob->ob.getBname())))
        .forEach(ob->System.out.println(ob));

        mappie.entrySet().stream()
        .sorted(Map.Entry.comparingByKey(Comparator.comparing(Books::getBname)))  //method reference
        .forEach(System.out::println); //method reference( instead of t -> System.out.println(t) )

        //reverse (need to cast if not using method references)
        mappie.entrySet().stream()
        .sorted(Map.Entry.comparingByKey(Comparator.comparing(ob->((Books) ob).getBname()).reversed()))
        .forEach(ob->System.out.println(ob));

        //or
        mappie.entrySet().stream().
        sorted(Map.Entry.comparingByKey( Comparator.comparing(Books::getBname).reversed() ))
        .forEach(System.out::println);
```

map() vs flatMap()

- Java 8 stream API provides map() and flatMap() method. Both these methods are **intermediate methods and returns another stream** as part of the output.
- Map() method used for transformation
- flatMap() method used for transformation + flattening



Data Transformation and Flattening

`Stream.of("a","b","c","d");`

[A, B, C, D]

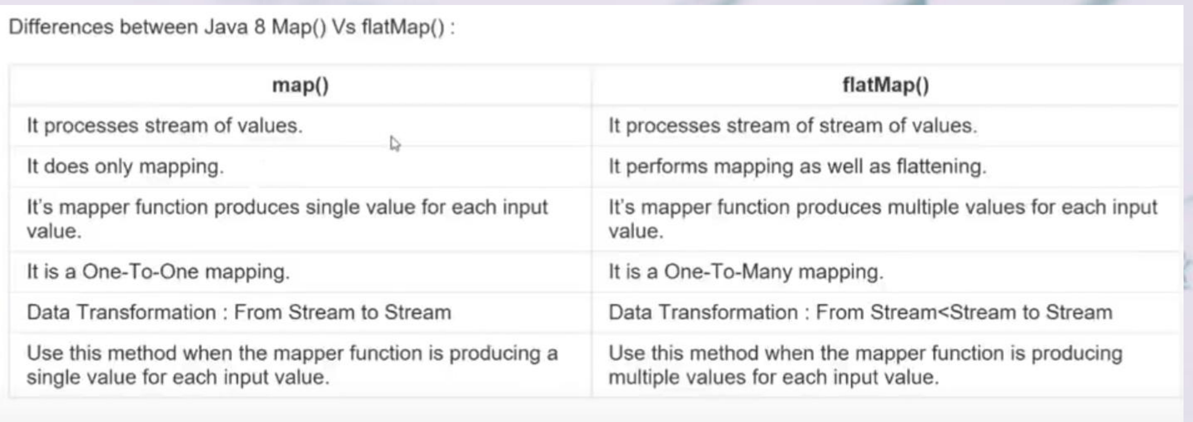
Transform data from lowercase to uppercase

`[[1, 2], [3, 4], [5, 6], [7, 8]]`

`[1, 2, 3, 4, 5, 6, 7, 8]`

convert stream of stream into single stream

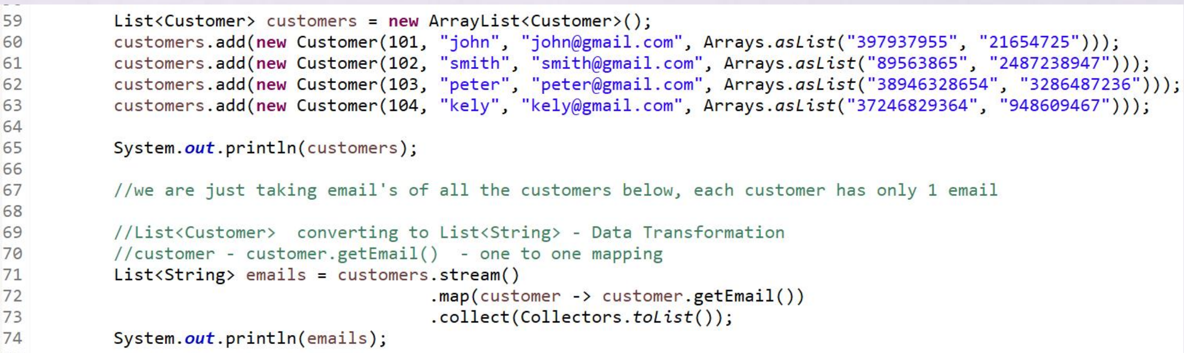
-
- Differences are:



Differences between Java 8 Map() Vs flatMap() :

map()	flatMap()
It processes stream of values.	It processes stream of stream of values.
It does only mapping.	It performs mapping as well as flattening.
It's mapper function produces single value for each input value.	It's mapper function produces multiple values for each input value.
It is a One-To-One mapping.	It is a One-To-Many mapping.
Data Transformation : From Stream to Stream	Data Transformation : From Stream<Stream to Stream
Use this method when the mapper function is producing a single value for each input value.	Use this method when the mapper function is producing multiple values for each input value.

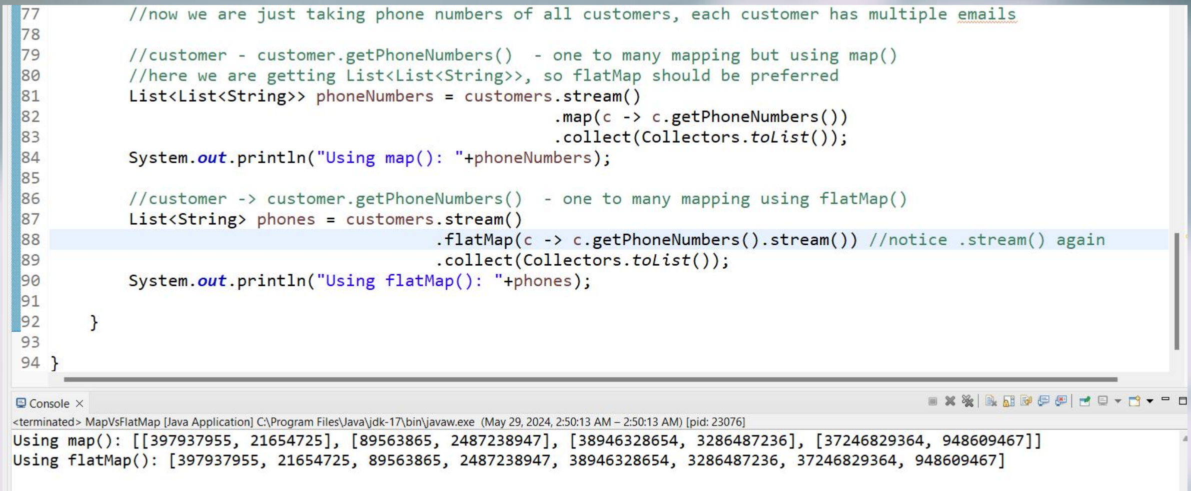
- Coding Example:
  If we have single result for each input value, then map() is preferred



```java
        List<Customer> customers = new ArrayList<Customer>();
        customers.add(new Customer(101, "john", "john@gmail.com", Arrays.asList("397937955", "21654725")));
        customers.add(new Customer(102, "smith", "smith@gmail.com", Arrays.asList("89563865", "2487238947")));
        customers.add(new Customer(103, "peter", "peter@gmail.com", Arrays.asList("38946328654", "3286487236")));
        customers.add(new Customer(104, "kely", "kely@gmail.com", Arrays.asList("37246829364", "948609467")));

        System.out.println(customers);

        //we are just taking email's of all the customers below, each customer has only 1 email

        //List<Customer>  converting to List<String> - Data Transformation
        //customer - customer.getEmail()  - one to one mapping
        List<String> emails = customers.stream()
                                 .map(customer -> customer.getEmail())
                                 .collect(Collectors.toList());
        System.out.println(emails);
```

If we have multiple result for each input value, then flatMap() is preferred, see the difference in output

(Also notice that we can use .collect(Collectors.toList()) to convert resulting stream to list)



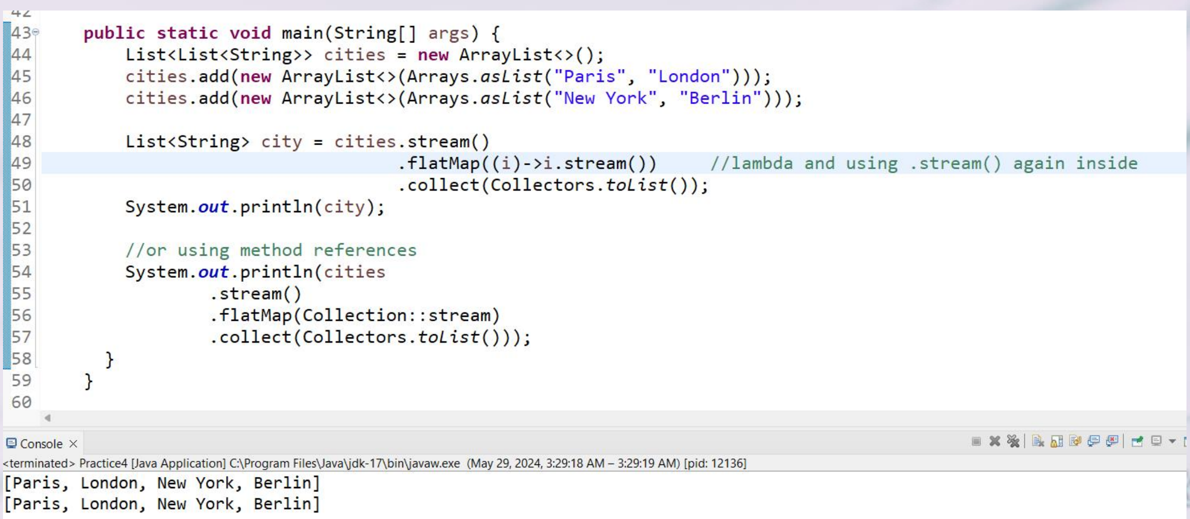
```java
        //now we are just taking phone numbers of all customers, each customer has multiple emails

        //customer - customer.getPhoneNumbers()  - one to many mapping but using map()
        //here we are getting List<List<String>>, so flatMap should be preferred
        List<List<String>> phoneNumbers = customers.stream()
                                                   .map(c -> c.getPhoneNumbers())
                                                   .collect(Collectors.toList());
        System.out.println("Using map(): "+phoneNumbers);

        //customer -> customer.getPhoneNumbers()  - one to many mapping using flatMap()
        List<String> phones = customers.stream()
                                       .flatMap(c -> c.getPhoneNumbers().stream()) //notice .stream() again
                                       .collect(Collectors.toList());
        System.out.println("Using flatMap(): "+phones);

    }

}
```

Console

<terminated> MapVsFlatMap [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (May 29, 2024, 2:50:13 AM – 2:50:13 AM) [pid: 23076]

```
Using map(): [[397937955, 21654725], [89563865, 2487238947], [38946328654, 3286487236], [37246829364, 948609467]]
Using flatMap(): [397937955, 21654725, 89563865, 2487238947, 38946328654, 3286487236, 37246829364, 948609467]
```

Another example:



```java
    public static void main(String[] args) {
        List<List<String>> cities = new ArrayList<>();
        cities.add(new ArrayList<>(Arrays.asList("Paris", "London")));
        cities.add(new ArrayList<>(Arrays.asList("New York", "Berlin")));

        List<String> city = cities.stream()
                                  .flatMap((i)->i.stream())     //lambda and using .stream() again inside
                                  .collect(Collectors.toList());
        System.out.println(city);

        //or using method references
        System.out.println(cities
                .stream()
                .flatMap(Collection::stream)
                .collect(Collectors.toList()));
      }
    }

```

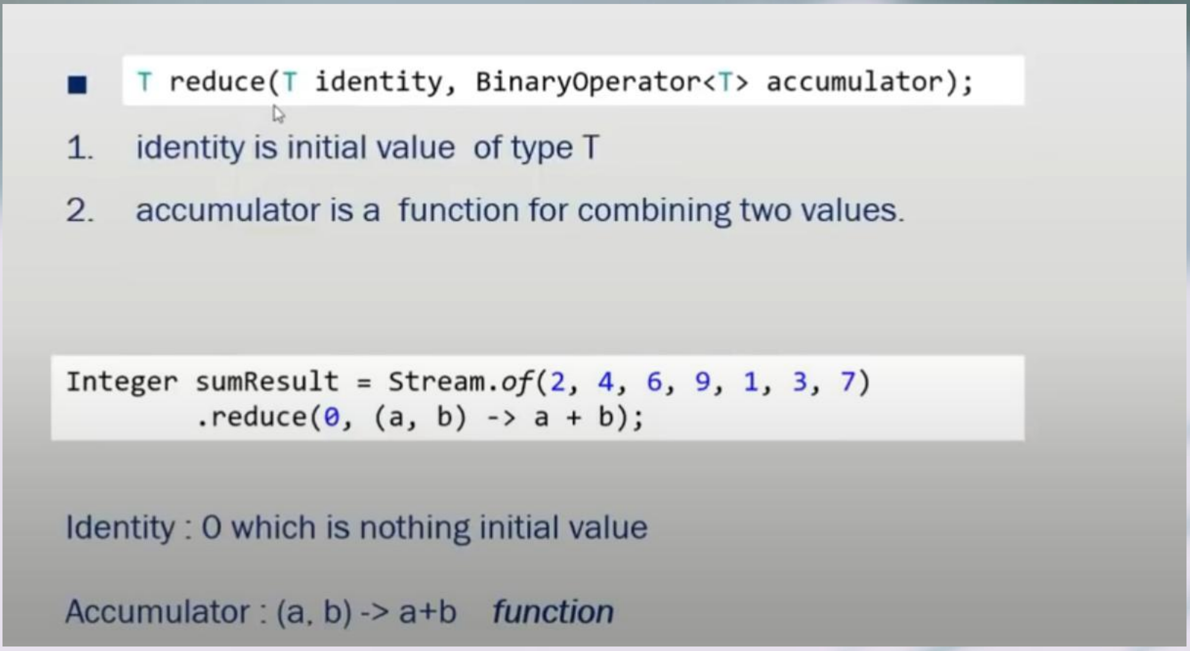
Console

<terminated> Practice4 [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (May 29, 2024, 3:29:18 AM – 3:29:19 AM) [pid: 12136]

```
[Paris, London, New York, Berlin]
[Paris, London, New York, Berlin]
```

Map() & Reduce()

- Map() as we know is an intermediate stream operator and is used for transforming data into another stream
- Reduce() is terminary stream operator and is used for reduction on the elements of stream using an associative accumulation function and **returns an Optional (not always)**
- 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
- For example,
  If our stream is like a list : [1, 3, 7, 4] and we have to find sum of numbers
  Then, map() - Transform the Stream<Object> to Stream of int
  Then, reduce() - Combine the above stream of int to produce the int which is the sum result
- It has reduce() method:



-
  ```
  T reduce(T identity, BinaryOperator<T> accumulator);
  ```

1. identity is initial value of type T
2. accumulator is a function for combining two values.

```
Integer sumResult = Stream.of(2, 4, 6, 9, 1, 3, 7)
        .reduce(0, (a, b) -> a + b);
```

Identity : 0 which is nothing initial value

Accumulator : (a, b) -> a+b *function*

- Coding example:

For finding sum of integers from a List<Integer>

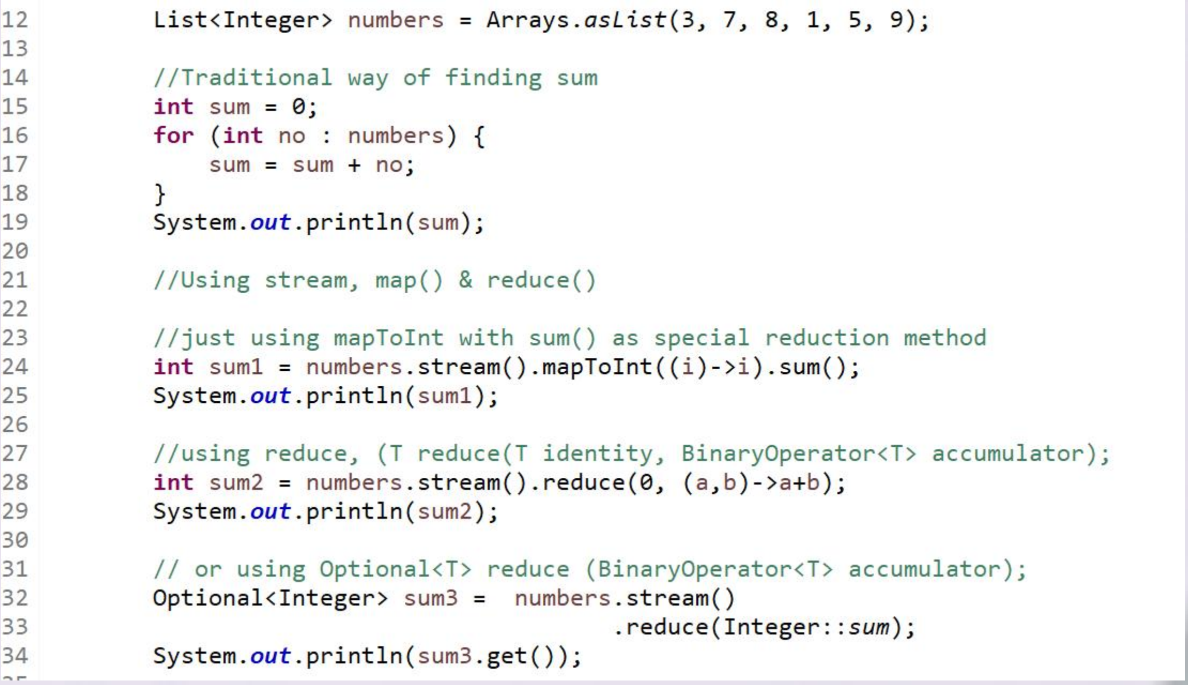
Note:

Both sum() and max() are reduce() operation methods. However, sum() needs int stream so we have to convert the list into int stream first using map() and then apply the reduce sum()

int sum = li.stream().mapToInt((i)->i).sum();

But for max(), we don't need to convert to int stream using map() as it uses Comparator.comparing which returns int stream in this case

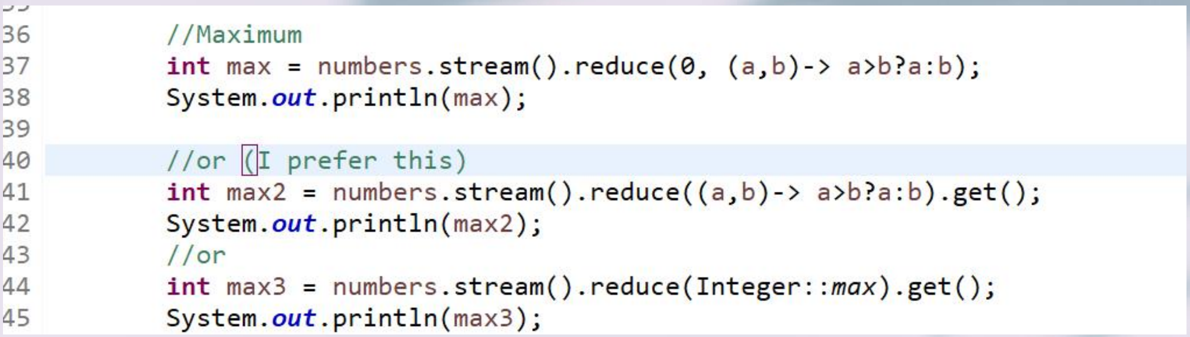
int max = li.stream().max(Comparator.comparing((i)->i)).get();



```
        List<Integer> numbers = Arrays.asList(3, 7, 8, 1, 5, 9);

        //Traditional way of finding sum
        int sum = 0;
        for (int no : numbers) {
            sum = sum + no;
        }
        System.out.println(sum);

        //Using stream, map() & reduce()

        //just using mapToInt with sum() as special reduction method
        int sum1 = numbers.stream().mapToInt((i)->i).sum();
        System.out.println(sum1);

        //using reduce, (T reduce(T identity, BinaryOperator<T> accumulator);
        int sum2 = numbers.stream().reduce(0, (a,b)->a+b);
        System.out.println(sum2);

        // or using Optional<T> reduce (BinaryOperator<T> accumulator);
        Optional<Integer> sum3 =  numbers.stream()
                                  .reduce(Integer::sum);
        System.out.println(sum3.get());
```

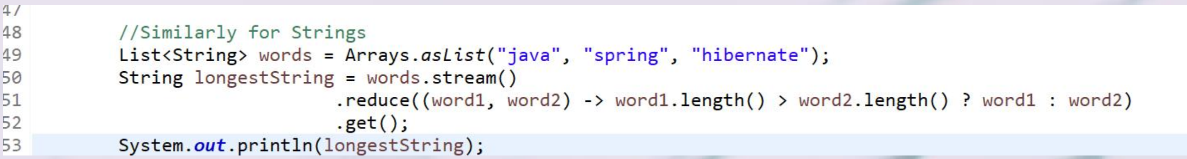
Similarly for finding maximum (or just literally use the max() method which is a reduce only)

```
int maxNumber = numbers.stream()
                                        .max(Comparator.comparing((i)->i))
                                        .get();
```



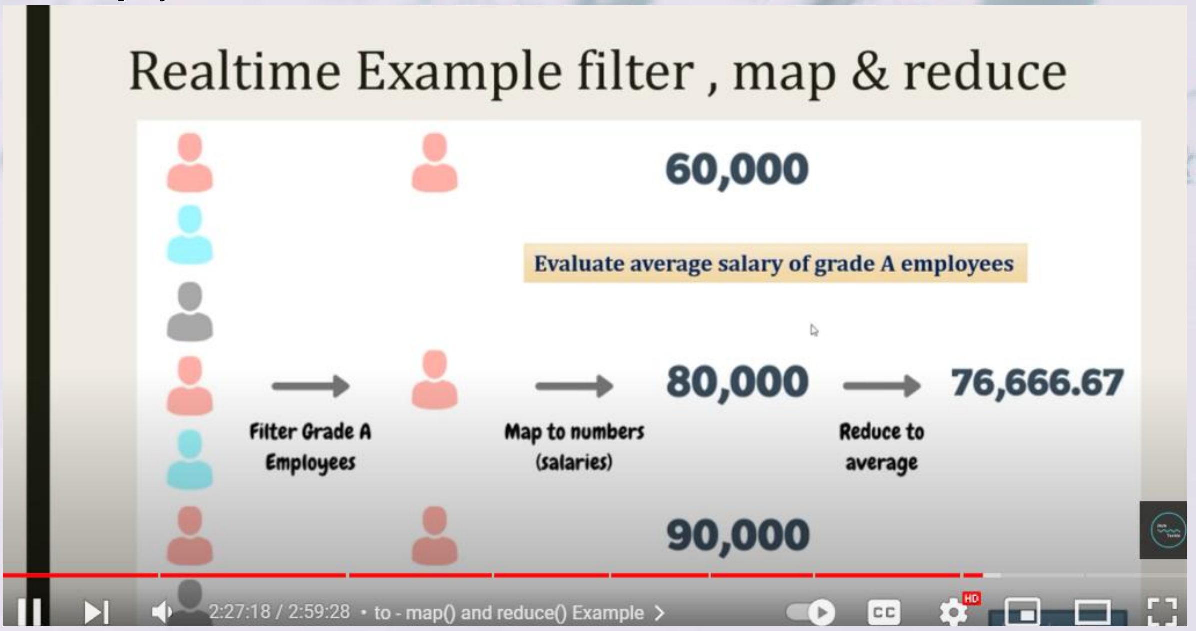
```
        //Maximum
        int max = numbers.stream().reduce(0, (a,b)-> a>b?a:b);
        System.out.println(max);

        //or (I prefer this)
        int max2 = numbers.stream().reduce((a,b)-> a>b?a:b).get();
        System.out.println(max2);
        //or
        int max3 = numbers.stream().reduce(Integer::max).get();
        System.out.println(max3);
```

Now, Similarly for Strings



```
        //Similarly for Strings
        List<String> words = Arrays.asList("java", "spring", "hibernate");
        String longestString = words.stream()
                            .reduce((word1, word2) -> word1.length() > word2.length() ? word1 : word2)
                            .get();
        System.out.println(longestString);
```

Now, suppose we want to do these operations in class object which has List<Employee>



Realtime Example filter , map & reduce

60,000

Evaluate average salary of grade A employees

80,000

76,666.67

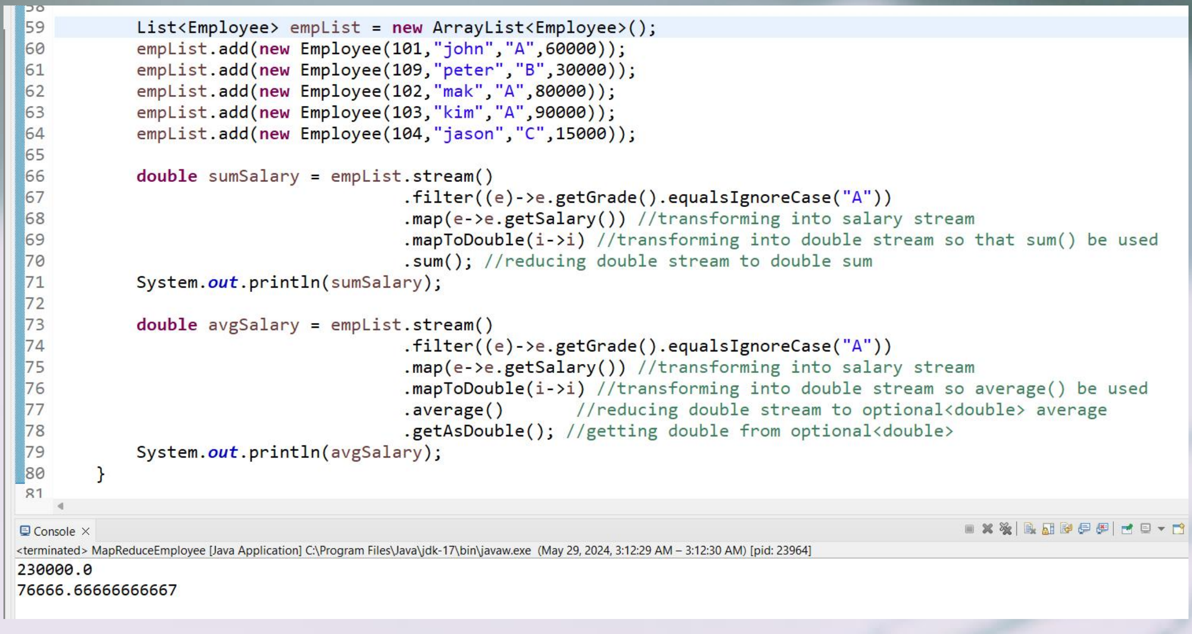
Filter Grade A Employees

Map to numbers (salaries)

Reduce to average

90,000

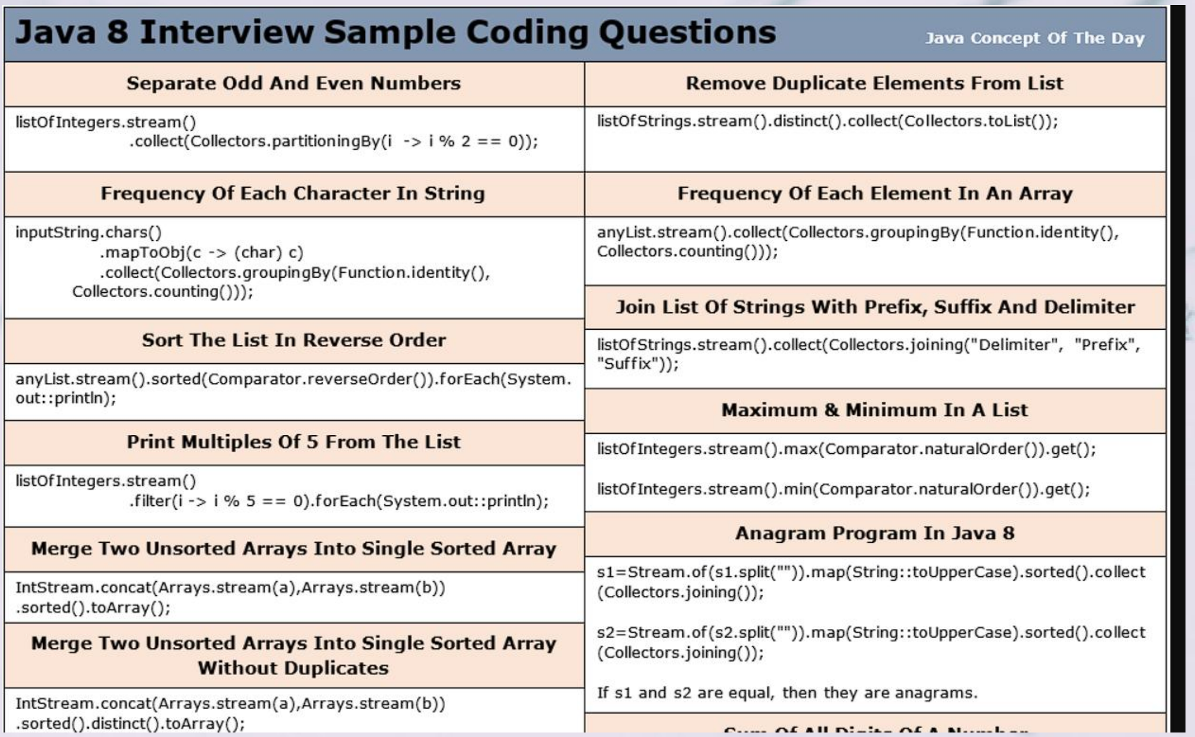
Then,



```
        List<Employee> empList = new ArrayList<Employee>();
        empList.add(new Employee(101,"john","A",60000));
        empList.add(new Employee(109,"peter","B",30000));
        empList.add(new Employee(102,"mak","A",80000));
        empList.add(new Employee(103,"kim","A",90000));
        empList.add(new Employee(104,"jason","C",15000));

        double sumSalary = empList.stream()
                                  .filter((e)->e.getGrade().equalsIgnoreCase("A"))
                                  .map(e->e.getSalary()) //transforming into salary stream
                                  .mapToDouble(i->i) //transforming into double stream so that sum() be used
                                  .sum(); //reducing double stream to double sum
        System.out.println(sumSalary);

        double avgSalary = empList.stream()
                                  .filter((e)->e.getGrade().equalsIgnoreCase("A"))
                                  .map(e->e.getSalary()) //transforming into salary stream
                                  .mapToDouble(i->i) //transforming into double stream so average() be used
                                  .average()       //reducing double stream to optional<double> average
                                  .getAsDouble(); //getting double from optional<double>
        System.out.println(avgSalary);
    }
```

```
<terminated> MapReduceEmployee [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe  (May 29, 2024, 3:12:29 AM – 3:12:30 AM) [pid: 23964]
230000.0
76666.66666666667
```

Some Java8 Sample Coding



Java 8 Interview Sample Coding Questions

Java Concept Of The Day

Separate Odd And Even Numbers

```
listOfIntegers.stream()
            .collect(Collectors.partitioningBy(i -> i % 2 == 0));
```

Frequency Of Each Character In String

```
inputString.chars()
        .mapToObj(c -> (char) c)
        .collect(Collectors.groupingBy(Function.identity(),
    Collectors.counting()));
```

Sort The List In Reverse Order

```
anyList.stream().sorted(Comparator.reverseOrder()).forEach(System.
out::println);
```

Print Multiples Of 5 From The List

```
listOfIntegers.stream()
            .filter(i -> i % 5 == 0).forEach(System.out::println);
```

Merge Two Unsorted Arrays Into Single Sorted Array

```
IntStream.concat(Arrays.stream(a),Arrays.stream(b))
.sorted().toArray();
```

Merge Two Unsorted Arrays Into Single Sorted Array Without Duplicates

```
IntStream.concat(Arrays.stream(a),Arrays.stream(b))
.sorted().distinct().toArray();
```

Remove Duplicate Elements From List

```
listOfStrings.stream().distinct().collect(Collectors.toList());
```

Frequency Of Each Element In An Array

```
anyList.stream().collect(Collectors.groupingBy(Function.identity(),
Collectors.counting()));
```

Join List Of Strings With Prefix, Suffix And Delimiter

```
listOfStrings.stream().collect(Collectors.joining("Delimiter", "Prefix",
"Suffix"));
```

Maximum & Minimum In A List

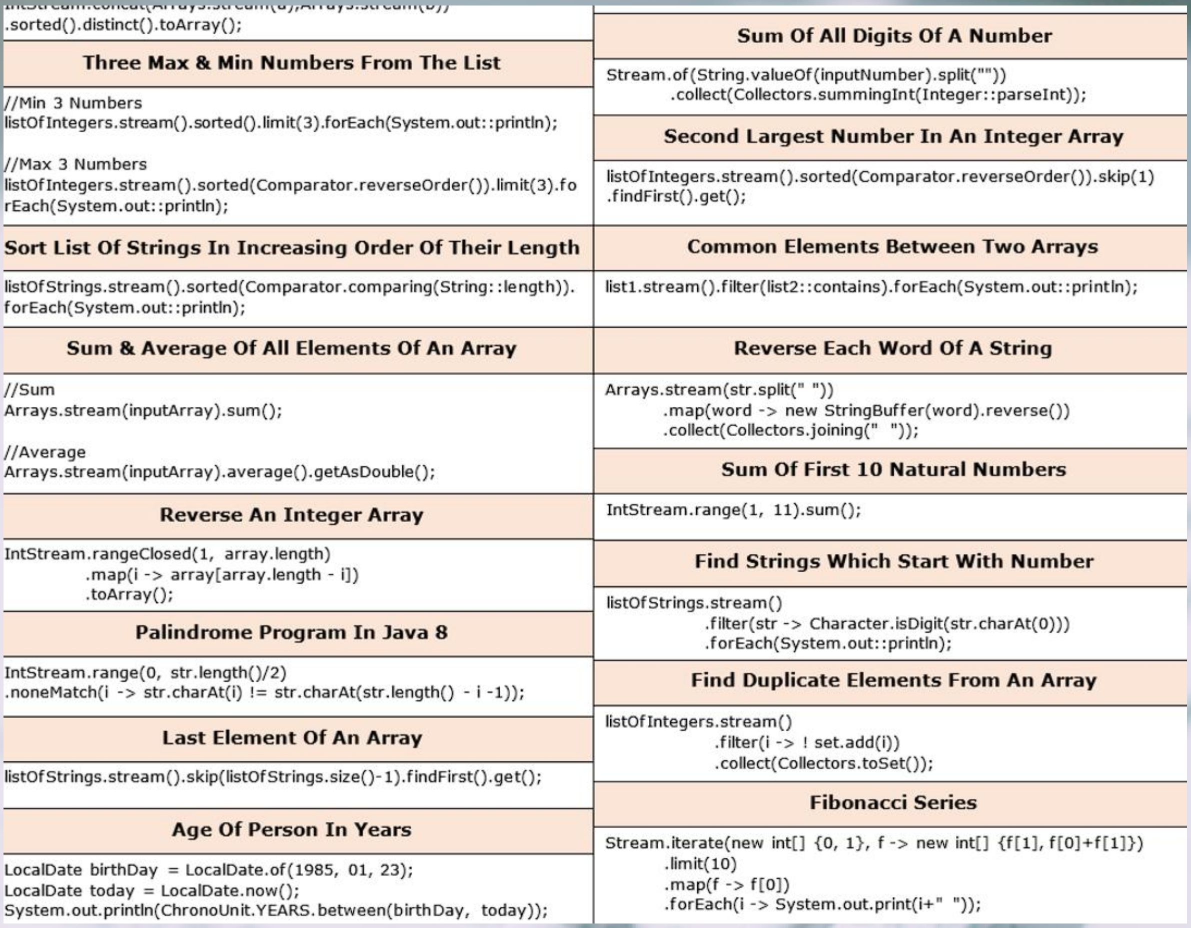
```
listOfIntegers.stream().max(Comparator.naturalOrder()).get();

listOfIntegers.stream().min(Comparator.naturalOrder()).get();
```

Anagram Program In Java 8

```
s1=Stream.of(s1.split("")).map(String::toUpperCase).sorted().collect
(Collectors.joining());

s2=Stream.of(s2.split("")).map(String::toUpperCase).sorted().collect
(Collectors.joining());
```

If s1 and s2 are equal, then they are anagrams.

Sum Of All Digits Of A Number



```
IntStream.concat(Arrays.stream(a),Arrays.stream(b))
.sorted().distinct().toArray();
```

Three Max & Min Numbers From The List

```
//Min 3 Numbers
listOfIntegers.stream().sorted().limit(3).forEach(System.out::println);

//Max 3 Numbers
listOfIntegers.stream().sorted(Comparator.reverseOrder()).limit(3).fo
rEach(System.out::println);
```

Sort List Of Strings In Increasing Order Of Their Length

```
listOfStrings.stream().sorted(Comparator.comparing(String::length)).
forEach(System.out::println);
```

Sum & Average Of All Elements Of An Array

```
//Sum
Arrays.stream(inputArray).sum();

//Average
Arrays.stream(inputArray).average().getAsDouble();
```

Reverse An Integer Array

```
IntStream.rangeClosed(1, array.length)
            .map(i -> array[array.length - i])
            .toArray();
```

Palindrome Program In Java 8

```
IntStream.range(0, str.length()/2)
.noneMatch(i -> str.charAt(i) != str.charAt(str.length() - i -1));
```

Last Element Of An Array

```
listOfStrings.stream().skip(listOfStrings.size()-1).findFirst().get();
```

Age Of Person In Years

```
LocalDate birthDay = LocalDate.of(1985, 01, 23);
LocalDate today = LocalDate.now();
System.out.println(ChronoUnit.YEARS.between(birthDay, today));
```

Sum Of All Digits Of A Number

```
Stream.of(String.valueOf(inputNumber).split(""))
        .collect(Collectors.summingInt(Integer::parseInt));
```

Second Largest Number In An Integer Array

```
listOfIntegers.stream().sorted(Comparator.reverseOrder()).skip(1)
.findFirst().get();
```

Common Elements Between Two Arrays

```
list1.stream().filter(list2::contains).forEach(System.out::println);
```

Reverse Each Word Of A String

```
Arrays.stream(str.split(" "))
        .map(word -> new StringBuffer(word).reverse())
        .collect(Collectors.joining(" "));
```

Sum Of First 10 Natural Numbers

```
IntStream.range(1, 11).sum();
```

Find Strings Which Start With Number

```
listOfStrings.stream()
              .filter(str -> Character.isDigit(str.charAt(0)))
              .forEach(System.out::println);
```

Find Duplicate Elements From An Array

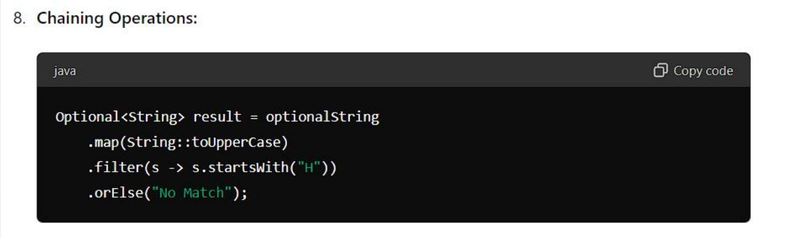
```
listOfIntegers.stream()
               .filter(i -> ! set.add(i))
               .collect(Collectors.toSet());
```

Fibonacci Series

```
Stream.iterate(new int[] {0, 1}, f -> new int[] {f[1], f[0]+f[1]})
        .limit(10)
        .map(f -> f[0])
        .forEach(i -> System.out.print(i+" "));
```

Optional Classes in Java8

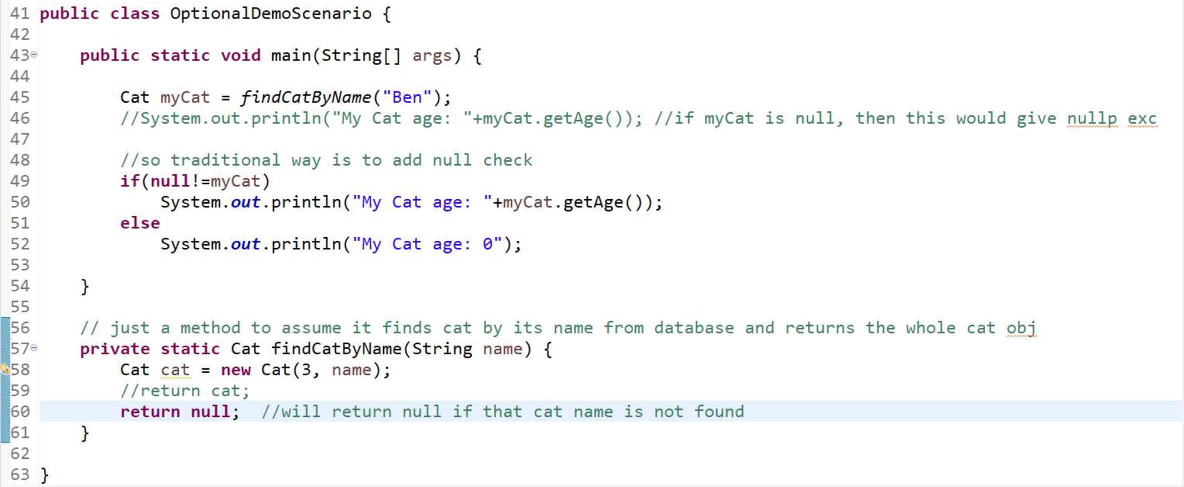
- Optional is a container object which may or may not contain a non-null value
- It can help in writing a neat code without using too many null checks
- It provides methods to handle the absence of values gracefully and perform operations in a functional style.
- It handles optional values without resorting to null references. It helps to avoid NullPointerException
- It is intended to use as **return** type, not everywhere where u have null variable scenario
- Optional provides methods like orElse and orElseGet to provide default values if the Optional is empty.
- Eg:



8. Chaining Operations:

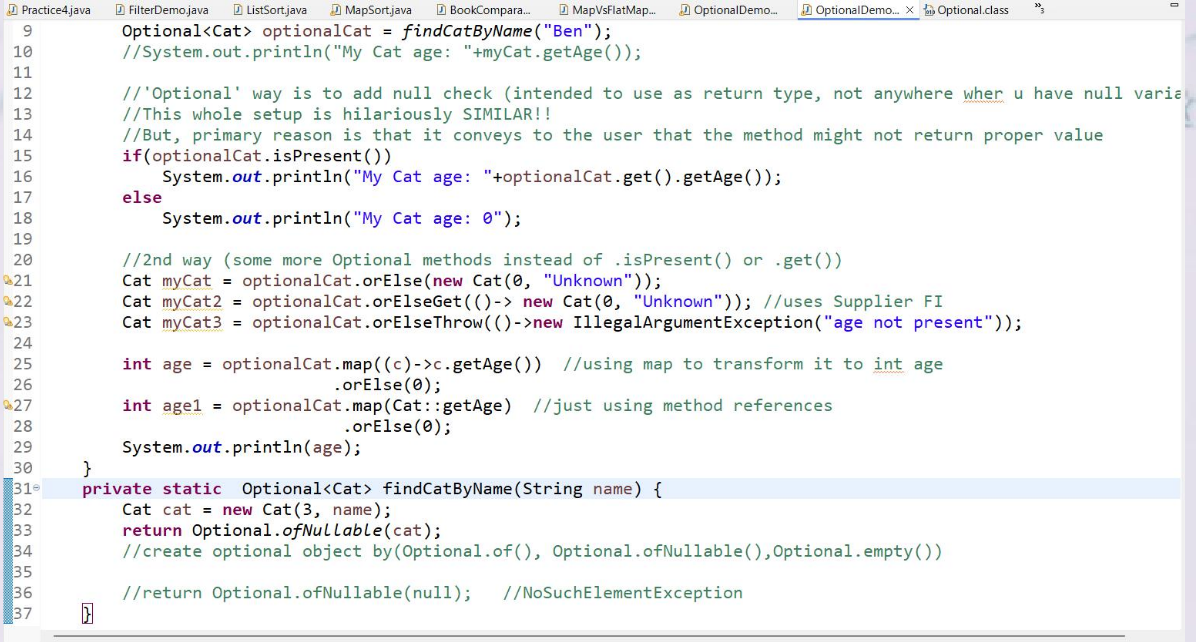
```java
Optional<String> result = optionalString
    .map(String::toUpperCase)
    .filter(s -> s.startsWith("H"))
    .orElse("No Match");
```

- Traditional way without Optional<>()



```java
public class OptionalDemoScenario {

    public static void main(String[] args) {

        Cat myCat = findCatByName("Ben");
        //System.out.println("My Cat age: "+myCat.getAge()); //if myCat is null, then this would give nullp exc

        //so traditional way is to add null check
        if(null!=myCat)
            System.out.println("My Cat age: "+myCat.getAge());
        else
            System.out.println("My Cat age: 0");

    }

    // just a method to assume it finds cat by its name from database and returns the whole cat obj
    private static Cat findCatByName(String name) {
        Cat cat = new Cat(3, name);
        //return cat;
        return null;  //will return null if that cat name is not found
    }

}
```

- By Using Optional class and it’s methods
  - Use these methods to create Optional object, Optional.of(), Optional.ofNullable(), Optional.empty()
  - Use optObj.isPresent() for checking & optObj.get() for getting
  - If optional element is null, we will get NoSuchElementException instead of NullPointerException
  - Use other methods like optObj.orElse() , optObj.orElseGet(), optObj.orElseThrow() etc
  - You can use optObj.map() for transformation, it returns <Optional> Integer below



```java
        Optional<Cat> optionalCat = findCatByName("Ben");
        //System.out.println("My Cat age: "+myCat.getAge());

        //'Optional' way is to add null check (intended to use as return type, not anywhere wher u have null varia
        //This whole setup is hilariously SIMILAR!!
        //But, primary reason is that it conveys to the user that the method might not return proper value
        if(optionalCat.isPresent())
            System.out.println("My Cat age: "+optionalCat.get().getAge());
        else
            System.out.println("My Cat age: 0");

        //2nd way (some more Optional methods instead of .isPresent() or .get())
        Cat myCat = optionalCat.orElse(new Cat(0, "Unknown"));
        Cat myCat2 = optionalCat.orElseGet(()-> new Cat(0, "Unknown")); //uses Supplier FI
        Cat myCat3 = optionalCat.orElseThrow(()->new IllegalArgumentException("age not present"));

        int age = optionalCat.map((c)->c.getAge())  //using map to transform it to int age
                             .orElse(0);
        int age1 = optionalCat.map(Cat::getAge)  //just using method references
                              .orElse(0);
        System.out.println(age);
    }
    private static  Optional<Cat> findCatByName(String name) {
        Cat cat = new Cat(3, name);
        return Optional.ofNullable(cat);
        //create optional object by(Optional.of(), Optional.ofNullable(),Optional.empty())

        //return Optional.ofNullable(null);   //NoSuchElementException
    }
```

Parallel Streams

- Java Parallel Streams is a feature of Java 8 and higher, meant for utilizing multiple cores of the processor.

- Normally any java code has one stream of processing, where it is executed sequentially. Whereas by using parallel streams, we can divide the code into multiple streams that are executed in parallel on separate cores and the final result is the combination of the individual outcomes.
- The order of execution, however, is not under our control.
- Therefore, it is advisable to use parallel streams in cases where no matter what is the order of execution, the result is unaffected and the state of one element does not affect the other as well as the source of the data also remains unaffected.