
Java™
8
(x) -> x + x

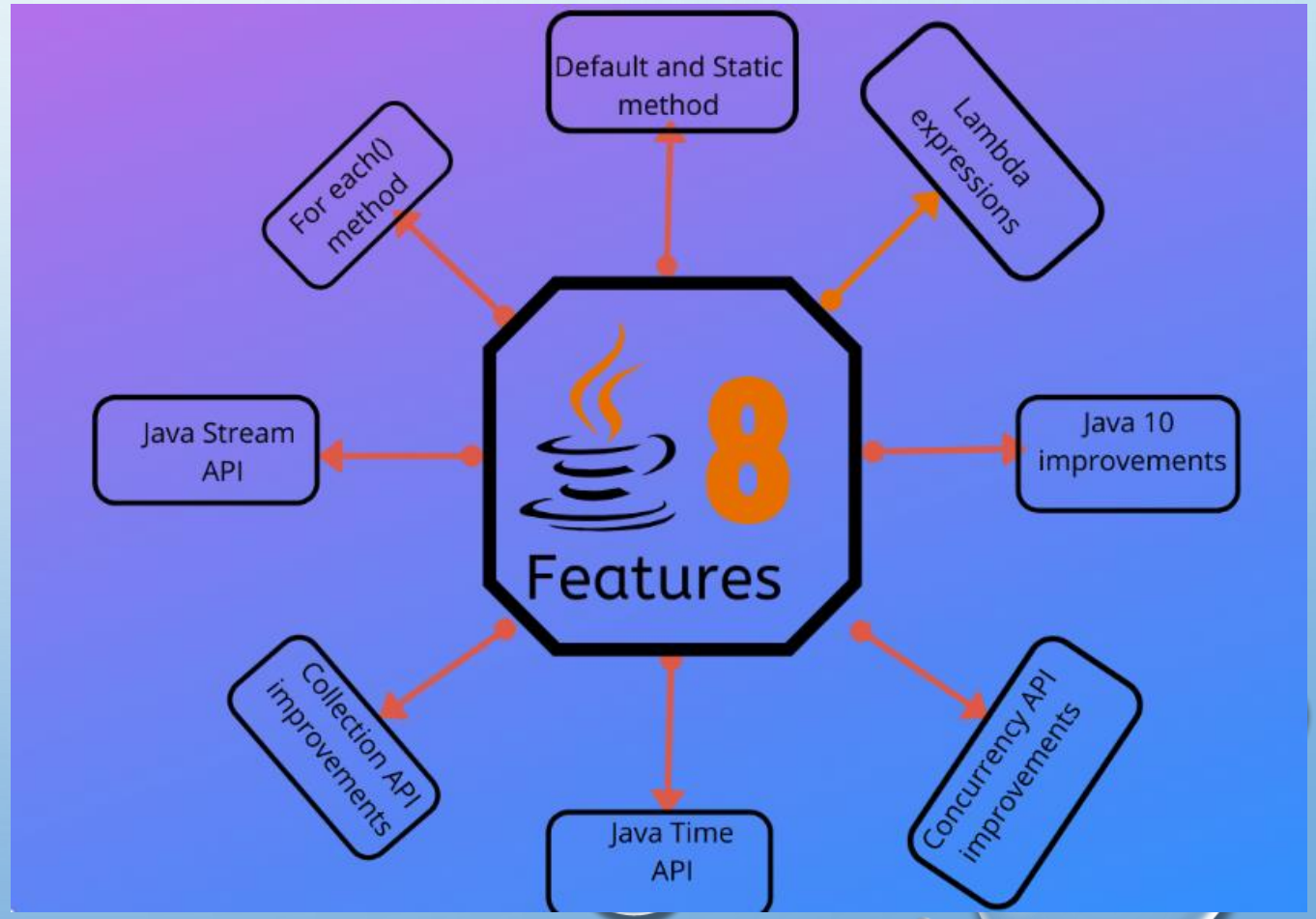
By Ayala Berkovich

JAVA 8

Java 8 was released on 18th March 2014. That's a long time ago but still many projects are running on Java 8.

Important **Java 8** features are:

1. **forEach()** method in **Iterable interface**
2. **default** and **static methods** in Interfaces
3. **Functional Interfaces** and **Lambda Expressions**
4. **Java Stream API** for Bulk Data Operations on Collections
5. **Java Time API**
6. **Collection API** improvements
7. **Concurrency API** improvements
8. **Java IO** improvements



JAVA COLLECTIONS

By Ayala Berkovich

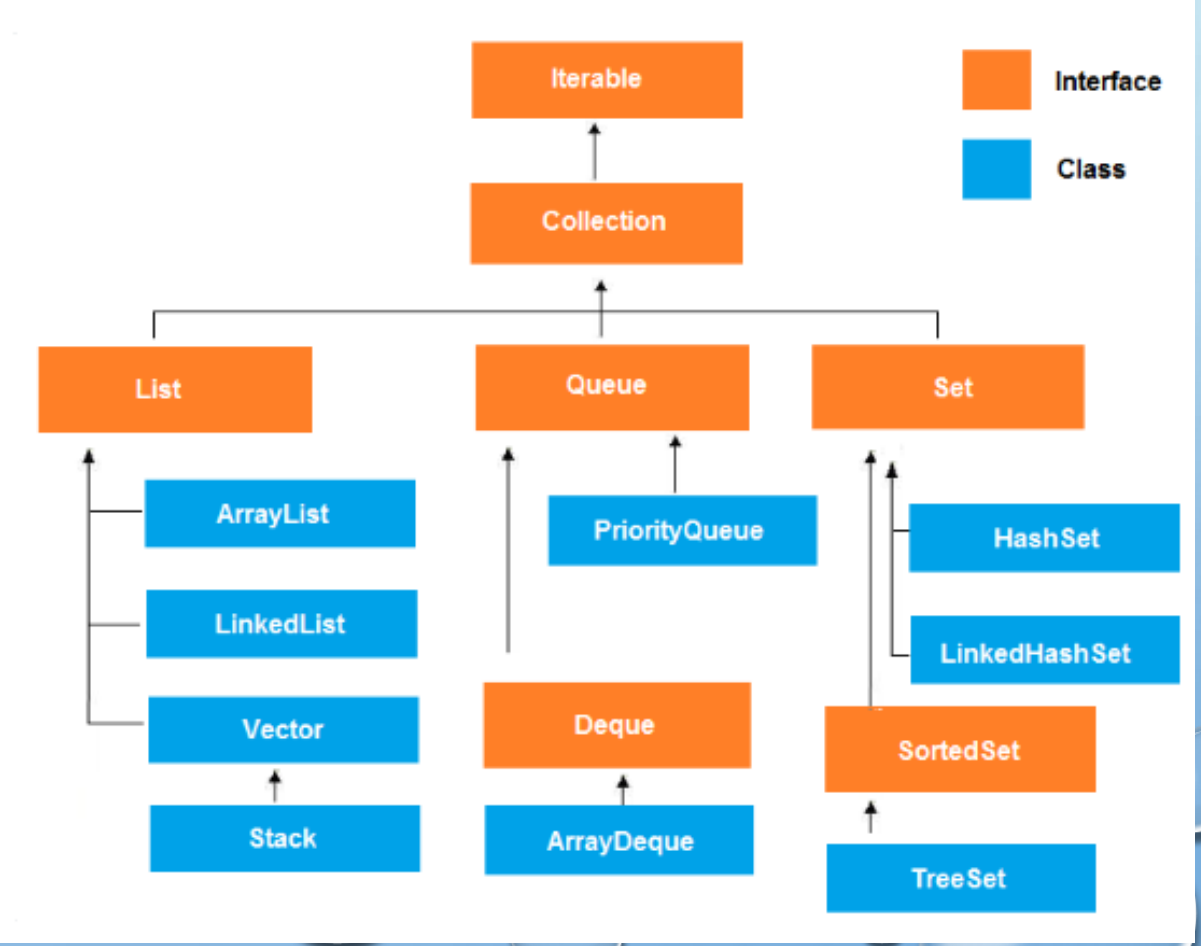
FOREACH() METHOD IN ITERABLE INTERFACE

Iterable is a super interface to **Collection**, so any class (such as Set or List…) that implements Collection also implements Iterable.
Has just one method:
Iterator<T> iterator()
Returns an iterator over a set of elements of type T

An `Iterator` is an object that can be used to loop through collections, like ArrayList and HashSet. It is called an "iterator" because "iterating" is the technical term for looping.
To use an Iterator, you must import it from the `java.util` package

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

    // Make a collection
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");

    // Get the iterator
    Iterator<String> it = cars.iterator();

    // Print the first item
    System.out.println(it.next());
  }
}
```

Method
public boolean hasNext()
public Object next()
public void remove()



JAVA 8 FEATURES

By Ayala Berkovich

The **forEach()** method in Java is a utility function to iterate over a Collection (list, set or map),
It is a default method defined in the **Iterable interface**.
Collection classes which extends **Iterable interface** can use **forEach** loop to iterate elements.
Let's see **forEach()** usage with a simple example.

```
public static void loopMapJava8() {
 Map<String, Integer> map = new HashMap<>();

map.put("A", 10);
map.put("B", 20);
map.put("C", 30);
map.put("D", 40);
map.put("E", 50);
map.put("F", 60);

map.forEach((k, v) -> System.out.println("Key : " + k + ", Value : " + v));
}
```

Output

```
Key : A, Value : 10
Key : B, Value : 20
Key : C, Value : 30
Key : D, Value : 40
Key : E, Value : 50
Key : F, Value : 60
```

By Ayala Berkovich

FOREACH() JAVA FEATURE

we can put logic operations inside foreach() statement:

```
public static void loopMapJava8() {
Map<String, Integer> map = new HashMap<>();

 map.put("A", 10);
 map.put("B", 20);
 map.put("C", 30);
 map.put(null, 40);
 map.put("E", null);
 map.put("F", 60);
 map.forEach( (k, v) -> {
                    if (k != null)
                        System.out.println("Key : " + k + ", Value : " + v);
                } } ); }
```

Output

```
Key : A, Value : 10
Key : B, Value : 20
Key : C, Value : 30
Key : E, Value : null
Key : F, Value : 60
```

By Ayala Berkovich

LAMBDA EXPRESSIONS JAVA FEATURE

A **lambda expression** is a short block of code which takes in parameters and returns a value.
Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.
simplest **lambda expression** contains a **single parameter** and an expression:

```
parameter -> expression
```

to use **more than one parameter,** wrap them in parentheses:

```
(parameter1, parameter2) -> expression
```

Expressions are limited and they can return value.
In order to do more complex operations code block can used with curly braces **{}** ,
if lambda expression needs to return a value, then the code block should have a **return statement**.

By Ayala Berkovich

LAMBDA EXPRESSIONS JAVA FEATURE

(argument-list) -> {body}

1) Argument-list: It can be empty or non-empty as well.
2) Body: It contains expressions and statements for lambda expression.

No Parameter Syntax

```
() -> {
//Body of no parameter lambda
}
```

By Ayala Berkovich

LAMBDA EXPRESSIONS JAVA FEATURE

```
public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(5);
    numbers.add(9);
    numbers.add(8);
    numbers.add(1);
    numbers.forEach( (n) -> { System.out.println(n); } );
  }
}
```

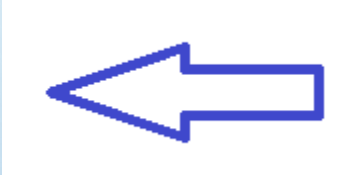
Output

```
5
9
8
1
```

By Ayala Berkovich

LAMBDA EXPRESSIONS JAVA FEATURE

```
interface StringFunction {
  String run(String str);
}

public class Main {
  public static void main(String[] args) {
    StringFunction exclaim = (s) -> s + "!";
    StringFunction ask = (s) -> s + "?";
    printFormatted("Hello", exclaim);
    printFormatted("Hello", ask);
  }
  public static void printFormatted(String str, StringFunction format) {
    String result = format.run(str);
    System.out.println(result);
  }
}
```



To use a **lambda expression in a method,** the method should have a parameter with a **single-method interface** as its type. **Calling the interface's method will run the lambda expression,**
method which takes a lambda expression as a parameter:

Lambda expressions can be stored in variables if the variable's type is an interface which has only one method.

The lambda expression should have the **same number of parameters** and the **same return type** as **that method.**


```
Hello!

Hello?
```

By Ayala Berkovich

LAMBDA EXPRESSIONS JAVA FEATURE

Lambda expression body consist block of code too:

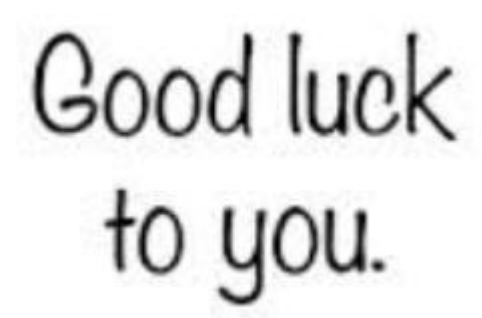
```
interface MyInterface
{double getPiValue();}

public class Main { public static void main( String[] args ) {
MyInterface MyInterface ref;
ref = () -> { double pi = 3.1415;
                        return pi;
              };
System.out.println("Value of Pi = " + ref.getPiValue());
  }
}
```

Output

Value of Pi =3.1415

By Ayala Berkovich

Write (use lambda expression) that gets text “smart_girls” and reverse it



By Ayala Berkovich

LAMBDA EXPRESSIONS JAVA FEATURE

```
interface MyInterface {
String reverse(String n);

}
public class MainTest
{ public static void main( String[] args )
   {
      MyInterface ref = (str) -> {
                        String result = "";
                        for (int i = str.length()-1; i >= 0 ; i--)
                            result += str.charAt(i);
                        return result; };

System.out.println("Lambda reversed = " +
ref.reverse("smart_girls"));
     }
}
```

Output

slrig_trams

By Ayala Berkovich

LAMBDA EXPRESSIONS JAVA FEATURE

Consumer Interface - is a part of the **java.util.function** package which has been introduced since Java 8

It represents a function which takes in **one argument** and produces a result.

Hence this functional interface which takes in one generic namely

•**T**: denotes the type of the input argument to the operation

```
public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(5);
    numbers.add(9);
    numbers.add(8);
    numbers.add(1);
    Consumer<Integer> method = (n) -> { System.out.println(n); };
    numbers.forEach( method );
  }
}
```

Output

```
5
9
8
1
```

CONSUMER INTERFACE

By Ayala Berkovich

The **lambda expression** assigned to an object of **Consumer type** is used to define its **accept()** which eventually applies the given operation on its argument.
Consumers are useful when it not needed to return any value as they are expected to operate via side-effects.

The Consumer interface consists of the following two functions:

```
void accept(T t)
```

And

```
default Consumer <T>
        andThen(Consumer<? super T> after)
```

CONSUMER INTERFACE - ACCEPT()

By Ayala Berkovich

```
void accept(T t)
```

Parameters: This method takes in one parameter:

- **t**– the input argument

Returns: This method does not return any value.

Below is the code to illustrate accept() method:

```java
public class Main {
    public static void main(String args[])
    {
        // Consumer to display a number
        Consumer<Integer> display = a -> System.out.println(a);

        // Implement display using accept()
        display.accept(10);

        // Consumer to multiply 2 to every integer of a list
        Consumer<List<Integer> > modify = list ->
        {
            for (int i = 0; i < list.size(); i++)
                list.set(i, 2 * list.get(i));
        };

        List<Integer> list = new ArrayList<Integer>();
        list.add(2);
        list.add(1);
        list.add(3);

        // Implement modify using accept()
        modify.accept(list);

    }
```

Output

```
10
4 2 6
```

CONSUMER INTERFACE - ACCEPT()

By Ayala Berkovich

```
default Consumer <T>
        andThen(Consumer<? super T> after)
```

Parameters: This method accepts a parameter **after** which is the Consumer to be applied after the current one.

Return Value: This method returns a composed Consumer that first applies the current Consumer first and then the after operation.

Exception: This method throws **NullPointerException** if the after operation is null.

Below is the code to illustrate andThen() method:

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

        // Consumer to multiply 2 to every integer of a list
        Consumer<List<Integer> > modify = list ->
        {
            for (int i = 0; i < list.size(); i++)
                list.set(i, 2 * list.get(i));
        };

        // Consumer to display a list of integers
        Consumer<List<Integer> >
            displist = list -> list.stream().forEach(a -> System.out.print(a + " "));

        List<Integer> list = new ArrayList<Integer>();
        list.add(2);
        list.add(1);
        list.add(3);

        // using addThen()
        modify.andThen(displist).accept(list);
        ;
    }
}
```

Output

```
4 2 6
```

JAVA 8 INTERFACE CHANGES

By Ayala Berkovich

default method in java interface

1.Java interface **default methods** break the differences between **interfaces** and **abstract classes**.
2.Java interface **default methods** will help us in removing base implementation classes, we can provide default implementation and the implementation classes can chose which one to override.
3.One of the major reason for introducing default methods in interfaces is to enhance the Collections API in Java 8 to support lambda expressions.
4.If any class in the hierarchy has a method with same signature, then default methods **become irrelevant**.
Java interface default methods are also referred to as Defender Methods or Virtual extension methods.

DEFAULT METHOD IN JAVA INTERFACE

By Ayala Berkovich

For creating a **default method** in java interface, we need to use "**default**" keyword with the method signature.

We know that Java doesn't allow us to extend multiple classes but can implements many interfaces:

```
public interface InterfaceA {

 default void printSomething() {
  System.out.println("I am inside A interface");
    }
}

public interface InterfaceB {

 default void printSomething() {
   System.out.println("I am inside B interface");
    }
}
```

```
public class Main implements InterfaceA, InterfaceB
{

}
```

If some class calls the printSomething() method from the object of Main class then which implementation will be called?

This class will not compile because of the Diamond problem in Java. (how system can know which method to use?)
To resolve the compilation issue, we will have to implement the printSomething() method as shown below:

DEFAULT METHOD IN JAVA INTERFACE

By Ayala Berkovich

```
public class Test implements InterfaceA, InterfaceB {

    @Override
    public void printSomething() {

      System.out.println("I am inside Main class");

      InterfaceA.super.printSomething();
      InterfaceB.super.printSomething();
    }

    public static void main(String args[]){
         Test test = new Test();
         test.printSomething();
    }
}
```

Output

```
I am inside Main class
I am inside A interface
I am inside B interface
```

STATIC METHOD IN JAVA INTERFACE

By Ayala Berkovich

The **static methods** in interfaces are similar to default methods but the **only difference is that you can't override them**. Now, why do we need static methods?If you don't want this implementation to be overridden in the implementing class…

```
public interface Vehicle {

 static void cleanVehicle(){
 System.out.println("I am cleaning vehicle");
    }
}

public class Car implements Vehicle {

    @Override
    public void cleanVehicle() {
     System.out.println("Cleaning the vehicle");
    }

    public static void main(String args[]) {
        Car car = new Car();
        car.cleanVehicle();

     }
}
```

If this code is good?

No, we will get compilation error in the Car class because a static method cannot be overridden!

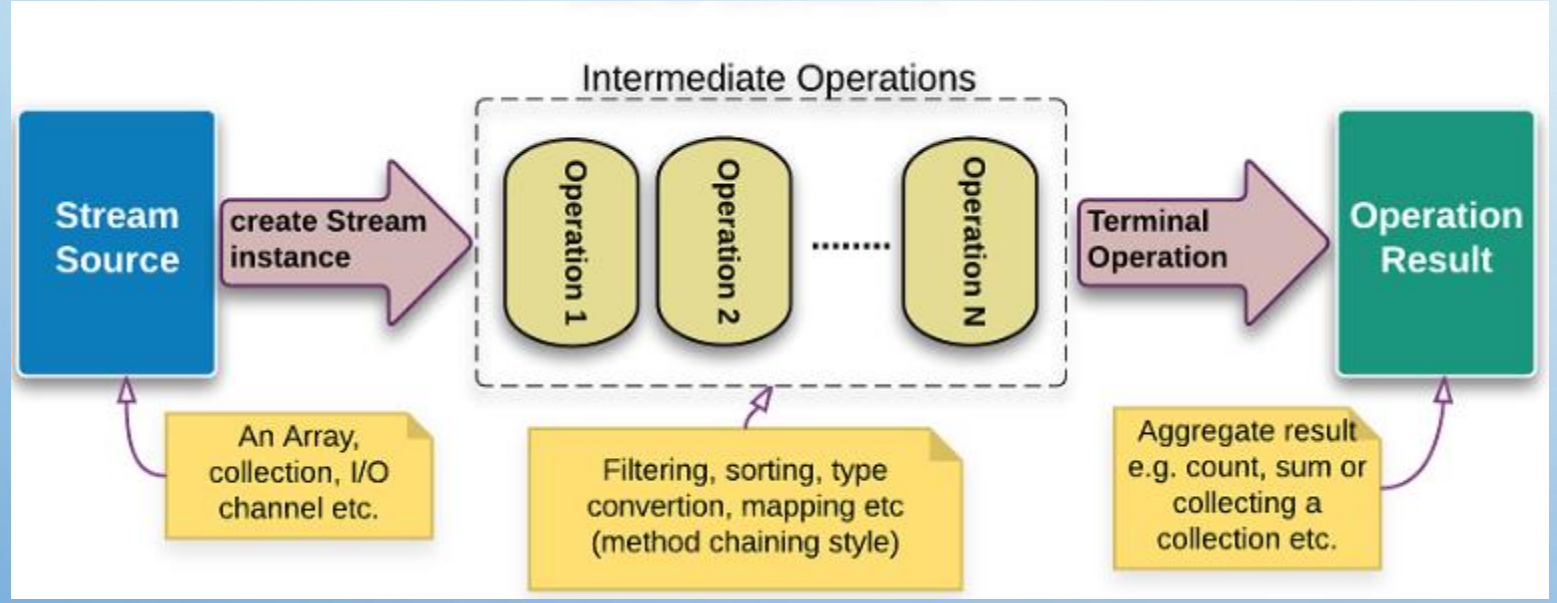
Good code is:

```
public class Car implements Vehicle {

    public static void main(String args[])
{
        Car car = new Car();

        Vehicle.cleanVehicle();
 }
}
```

By Ayala Berkovich

JAVA STREAM API

Using **stream**, you can process data in a declarative way (in which programs describe their desired results without explicitly listing commands or steps that must be performed.) similar to SQL statements:



Like in : **SELECT max(salary),employee_name from Employee**

In this SQL expression automatically returns the maximum salaried employee's details, without doing any computation

By Ayala Berkovich

JAVA STREAM API

How to create stream?

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

import java.util.stream.Stream;

public class Test {

    public static void main(String[] args) {

        List<Integer> num = Arrays.asList(5,3,6,7,8,9);
        //create stream
        Stream<Integer> data = num.stream();
        data.forEach(n->System.out.println(n));

    }

}
```

But why we need it?

In this way we work **just on stream** and real data will not be changed!
But we can use stream just ones...you can't reuse it..
Example:

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

import java.util.stream.Stream;

public class Test {

    public static void main(String[] args) {

        List<Integer> num = Arrays.asList(5,3,6,7,8,9);
        //create stream
        Stream<Integer> data = num.stream();
        data.forEach(n->System.out.println(n));
        //use stream more than ones
        data.forEach(n->System.out.println(n));
```

```
5
3
6
7
8
9
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated
        at java.base/java.util.stream.AbstractPipeline.sourceStageSpliterator(AbstractPipeli
        at java.base/java.util.stream.ReferencePipeline$Head.forEach(ReferencePipeline.java:
        at For_test/test.Test.main(Test.java:17)
```

By Ayala Berkovich

JAVA STREAM API

Stream operations:

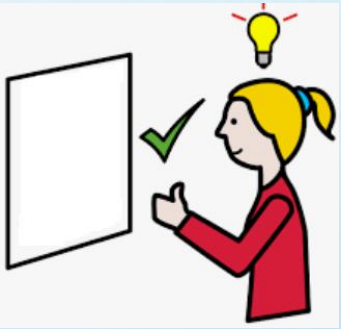
forEach()
map() - is used to map each element to its corresponding result **map(n->n*2));**
filter() - is used to eliminate elements based on a criteria **filter(string->string.isEmpty());**
limit() - is used to reduce the size of the stream
sorted() - is used to sort the stream
count() – is used to count elements in the stream
....

Output of all this operations is **Stream!**

By Ayala Berkovich

JAVA STREAM API

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

    List<Integer> num = Arrays.asList(5,3,6,7,8,9);
    //1. create stream
    Stream<Integer> data = num.stream();
    data.forEach(n->System.out.println(n));
    //use stream more than ones
    //data.forEach(n->System.out.println(n));
    //2. sort values
    Stream<Integer> sortedData = num.stream().sorted();
    sortedData.forEach(n->System.out.println(n));
    System.out.println("_________________\n");

    //3.map values

    Stream<Integer> datamap = num.stream().sorted().map(n->n*2);
    datamap.forEach(n->System.out.println(n));

    System.out.println("_________________\n");

    //4.filter

    Stream<Integer> datafilter = num.stream().filter(n->n%2==1).map(n->n*2).sorted();

    datafilter.forEach(n->System.out.println(n));

}
```



Output

```
5
3
6
7
8
9
3
5
6
7
8
9
_____

6
10
12
14
16
18
_____

6
10
14
18
```

By Ayala Berkovich

JAVA STREAM API

```
public class StreamApplication {
 public static void main(String[] args)
 {
   List<String> list = Arrays.asList("9", "A", "Z", "1", "B", "Y", "4", "a", "c");

   List<String> sortedList = list.stream().sorted().collect(Collectors.toList());
sortedList.forEach(System.out::println);
 }
}
```

Output

```
1
4
9
A
B
Y
Z
a
c
```

By Ayala Berkovich

JAVA STREAM API

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

List<String> list = Arrays.asList("9", "A", "Z", "1", "B", "Y", "4", "a", "c");

List<String> sortedList = list.stream().sorted(Comparator.reverseOrder())
.collect(Collectors.toList()); sortedList.forEach(System.out::println);
 }
```

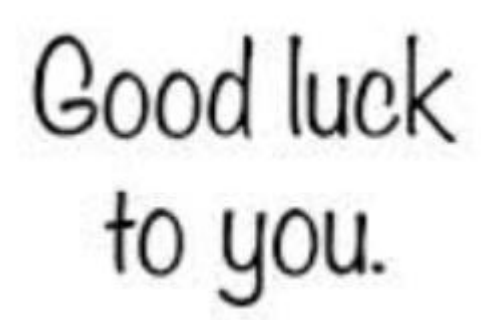
Output

```
c
a
Z
Y
B
A
9
4
1
```

By Ayala Berkovich

JAVA STREAM API

Write code that take 10 random integer values ,sort it, takes only odd values and print values only distinct values.



By Ayala Berkovich

JAVA STREAM API

```
Random random = new Random();
int upperbound = 20;
 // Generating random values from 0 - 20
List<Integer> numRandom = new ArrayList<Integer>();
 for(int i = 0; i < 10; i++) {
   // System.out.println(random.nextInt(upperbound));
    numRandom.add(random.nextInt(upperbound));
 }

 Stream<Integer> st = numRandom.stream().sorted().filter(n->n%2==0).distinct();
 st.forEach(n->System.out.println(n));
```

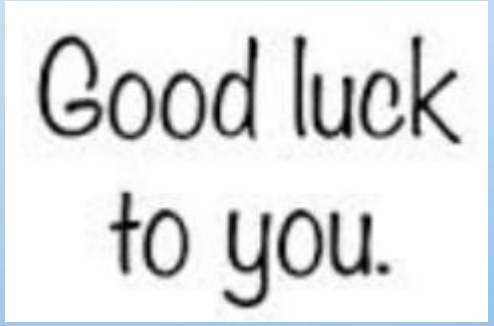
```
0
4
8
16
```

By Ayala Berkovich

JAVA STREAM API

Write code that defined List of names: List of names: (“Avraam”,”Sara”,”Izhak”,”Rivka”,”Yaakov”,”Lea”,””,”Rachel”) ,operate on no empty string only :

- Get just names with name size more than 4.
- Output them with”;” behind the names



By Ayala Berkovich

JAVA STREAM API

```
List<String> names = Arrays.asList("Avraam","Sara","Izhak","Rivka","Yaakov","Lea","","Rachel");
Stream str = names.stream();
 str.forEach(n->System.out.println(n));
String str2 = names.stream().filter(string ->!string.isEmpty()).collect(Collectors.joining("; "));
System.out.println(str2);
```

Collectors in java 8 help accumulate the Stream's elements in the form of data structures;

```
Avraam
Sara
Izhak
Rivka
Yaakov
Lea


Avraam; Sara; Izhak; Rivka; Yaakov; Lea; Rachel
```

By Ayala Berkovich

JAVA STREAM API

More example of use Collectors:

```
Stream<Student> stream = Stream.of(
  new Student(1231L, "Strong", "Belwas"),
  new Student(42324L, "Barristan", "Selmy"),
  new Student(15242L, "Arthur", "Dayne")
);

Map<Long, String> map = stream.collect(Collectors
    .toMap(Student::getId, Student::getFirstName));

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

Output

```
{42324=Barristan, 15242=Arthur, 1231=Strong}
```

By Ayala Berkovich

COLLECTION API IMPROVEMENTS

Legacy code for checking **containsKey()** moved to default method **getOrDefault().**
This method returns the value to which the specified key is mapped, otherwise returns the given defaultValue if this map contains no mapping for the key:

```
default V getOrDefault(Object key, V defaultValue)
```

Parameters: This method accepts two parameters:
- **key:** which is the **key** of the element whose value has to be obtained.
- **defaultValue:** which is the **default value** that has to be returned, if no value is mapped with the specified key.

Return Value: This method returns **value** mapped with the specified key, otherwise **default value** is returned.

```
Map<String, String> map = new HashMap<>();
map.put("C", "c");
String val = map.getOrDefault("B", "Test!");
System.out.println(val);
```

Output

```
Test
```

COLLECTION API IMPROVEMENTS

By Ayala Berkovich

Replace utilities:

replaceAll() can replace all the values in a single attempt

replace(K key,V oldValue,V newValue) method replaces the the entry for the specified key only if currently mapped to specified value

```
Map<String, String> map = new HashMap<>();
map.put("C", "c");
map.put("B", "b");
map.replaceAll((k, v) -> "x”);
```

Output

```
values is "x" for all keys.
```

putIfAbsent – put value if no exist

```
Map<String, String> map = new HashMap<>();
map.put("C", "c");
map.put("B", "b");
map.putIfAbsent("B", "x");
System.out.println(map.get("B"));
```

Output

```
prints "b"
```

```
If key “B” not exist
will add this element
with default value ”x”
```

By Ayala Berkovich

COLLECTION API IMPROVEMENTS

compute(),computeIfPresent(),computeIfAbsent() - needed to get the value for specific keys, process it and put them back

```
Map<String, String> map = new HashMap<>();
map.put("C", "c");
map.put("B", "b");
map.compute("B", (k, v) -> v.concat(" - new "));
System.out.println(map.get("B"));
```

prints "b - new"

merge() - more useful when you are combining maps or appending values for duplicated keys.

```
Map<String, String> map = new HashMap<>();
map.put("C", "c");
map.put("B", "b");
map.merge("B", "Test", (v1, v2) -> v1 + v2);
System.out.println(map.get("B"));
```

prints "bTest"

JAVA TIME API

By Ayala Berkovich

```java
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.format.DateTimeFormatter;

public class Zone {

// Function to get Zoned Date and Time
public static void ZonedTimeAndDate()
{
    LocalDateTime date = LocalDateTime.now();
    DateTimeFormatter format1 =
      DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

    String formattedCurrentDate = date.format(format1);

    System.out.println("formatted current Date and"+
                       " Time : "+formattedCurrentDate);

    // to get the current zone
    ZonedDateTime currentZone = ZonedDateTime.now();
    System.out.println("the current zone is "+
                        currentZone.getZone());

    // getting time zone of specific place
    // we use withZoneSameInstant(): it is
    // used to return a copy of this date-time
    // with a different time-zone,
    // retaining the instant.
    ZoneId tokyo = ZoneId.of("Asia/Tokyo");

    ZonedDateTime tokyoZone =
            currentZone.withZoneSameInstant(tokyo);

    System.out.println("tokyo time zone is " +
                        tokyoZone);
```

Java 8 under the package **java.time** introduced a new date-time API, most important classes :

1. Zoned : Specialized date-time API to deal with various timezones.

Output

```
formatted current Date and Time : 09-04-2018 06:21:13
the current zone is Etc/UTC
tokyo time zone is 2018-04-09T15:21:13.220+09:00[Asia/Tokyo]
```

By Ayala Berkovich

JAVA TIME API

```java
import java.time.*;
import java.time.format.DateTimeFormatter;

public class Date {

public static void LocalDateTimeApi()
{

    // the current date
    LocalDate date = LocalDate.now();
    System.out.println("the current date is "+
                        date);


    // the current time
    LocalTime time = LocalTime.now();
    System.out.println("the current time is "+
                        time);


    // will give us the current time and date
    LocalDateTime current = LocalDateTime.now();
    System.out.println("current date and time : "+
                        current);


    // to print in a particular format
    DateTimeFormatter format =
      DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

    String formatedDateTime = current.format(format);

    System.out.println("in formatted manner "+
                        formatedDateTime);
```

Local : Simplified date-time API with no complexity of timezone handling

- **LocalDate/LocatTime** and **LocalDateTime API :** Use it when time zones are NOT required.

Output

```
the current date is 2021-09-23
the current time is 20:52:39.954238
current date and time : 2021-09-23T20:52:39.956909
in formatted manner 23-09-2021 20:52:39
```

By Ayala Berkovich