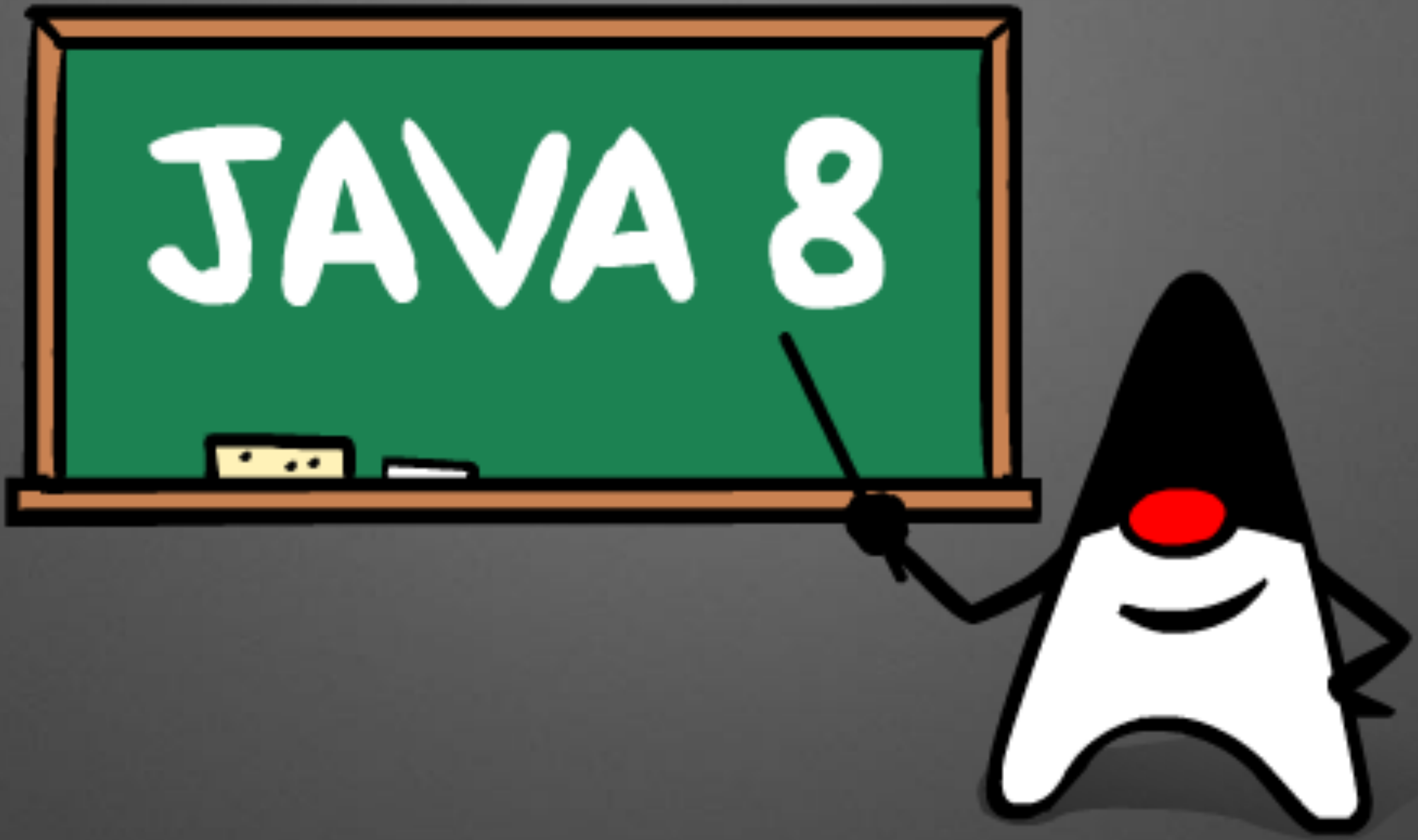


Cengiz Bursalıoğlu

Distribution and CRM Development
Project Specialist

Introduction to Java 8

JAVA 8 is a major feature release of JAVA programming language development. Its initial version was released on 18 March 2014.

With the Java 8 release, Java provided supports for functional programming, new JavaScript engine, new APIs for date time manipulation, new streaming API, etc.

Java 8 New Features

- **Lambda expression** – Adds functional processing capability to Java.
- **Method references** – Referencing functions by their names instead of invoking them directly. Using functions as parameter.
- **Default method** – Interface to have default method implementation.
- **Stream API** – New stream API to facilitate pipeline processing.
- **Date Time API** – Improved date time API.
- **Optional** – Emphasis on best practices to handle null values properly.
- **Nashorn, JavaScript Engine** – A Java-based engine to execute JavaScript code.

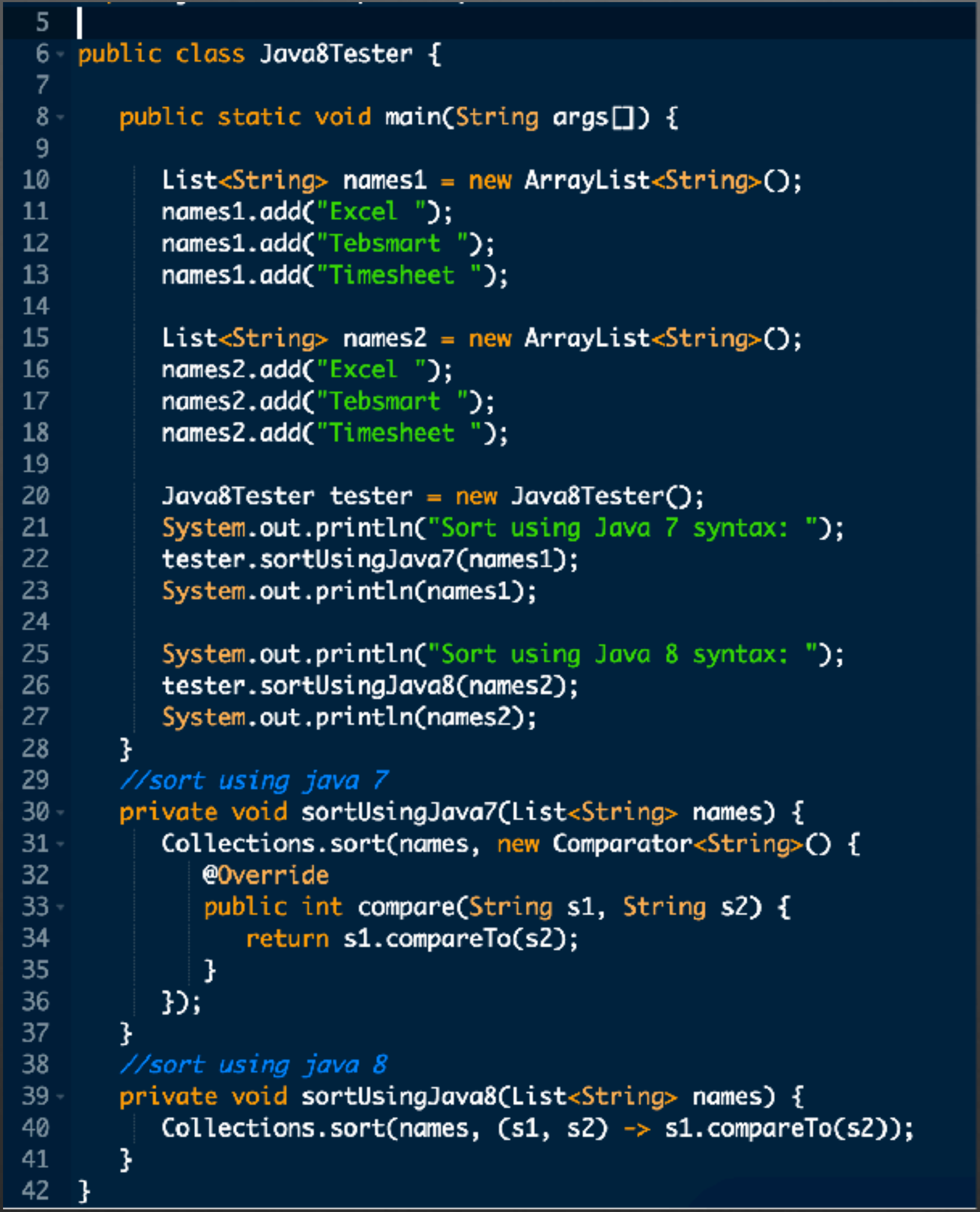
Lambda Expressions

Lambda expressions are introduced in Java 8 and are touted to be the biggest feature of Java 8. Lambda expression facilitates functional programming, and simplifies the development a lot.

A lambda expression is characterized by the following syntax.

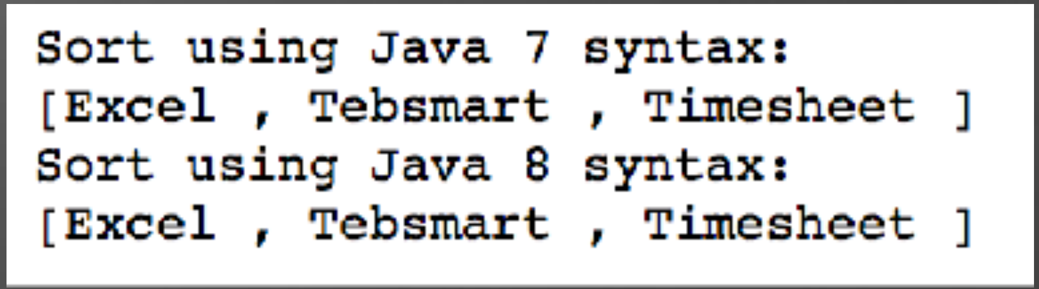
- parameter -> expression body

go to the next page to see examples

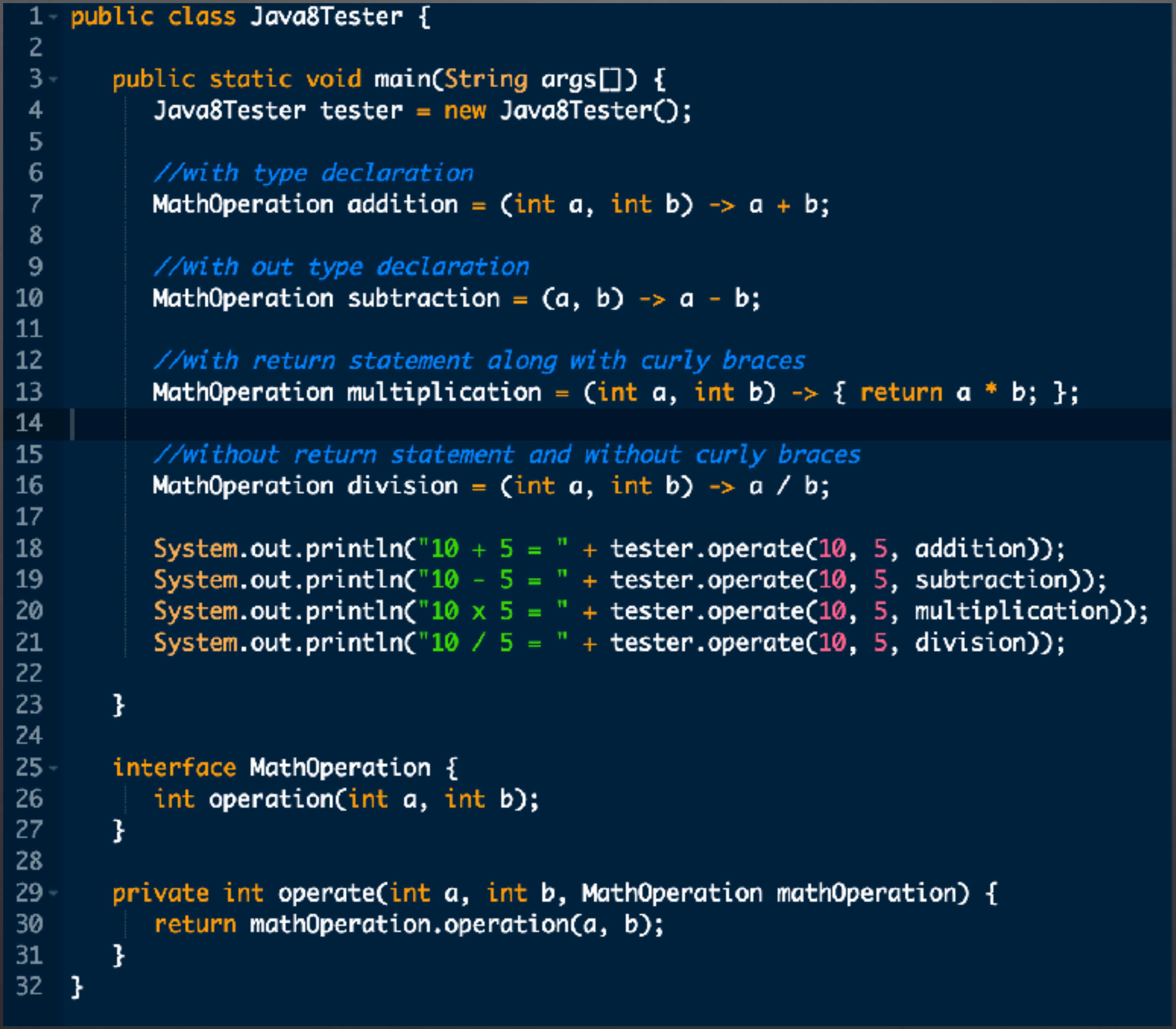


```java
public class Java8Tester {

    public static void main(String args[]) {

        List<String> names1 = new ArrayList<String>();
        names1.add("Excel ");
        names1.add("Tebsmart ");
        names1.add("Timesheet ");

        List<String> names2 = new ArrayList<String>();
        names2.add("Excel ");
        names2.add("Tebsmart ");
        names2.add("Timesheet ");

        Java8Tester tester = new Java8Tester();
        System.out.println("Sort using Java 7 syntax: ");
        tester.sortUsingJava7(names1);
        System.out.println(names1);

        System.out.println("Sort using Java 8 syntax: ");
        tester.sortUsingJava8(names2);
        System.out.println(names2);
    }
    //sort using java 7
    private void sortUsingJava7(List<String> names) {
        Collections.sort(names, new Comparator<String>() {
            @Override
            public int compare(String s1, String s2) {
                return s1.compareTo(s2);
            }
        });
    }
    //sort using java 8
    private void sortUsingJava8(List<String> names) {
        Collections.sort(names, (s1, s2) -> s1.compareTo(s2));
    }
}
```

Output:

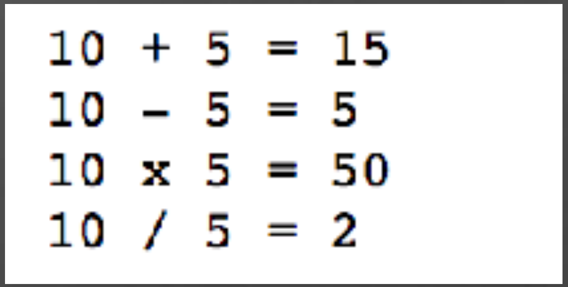


```
Sort using Java 7 syntax:
[Excel , Tebsmart , Timesheet ]
Sort using Java 8 syntax:
[Excel , Tebsmart , Timesheet ]
```

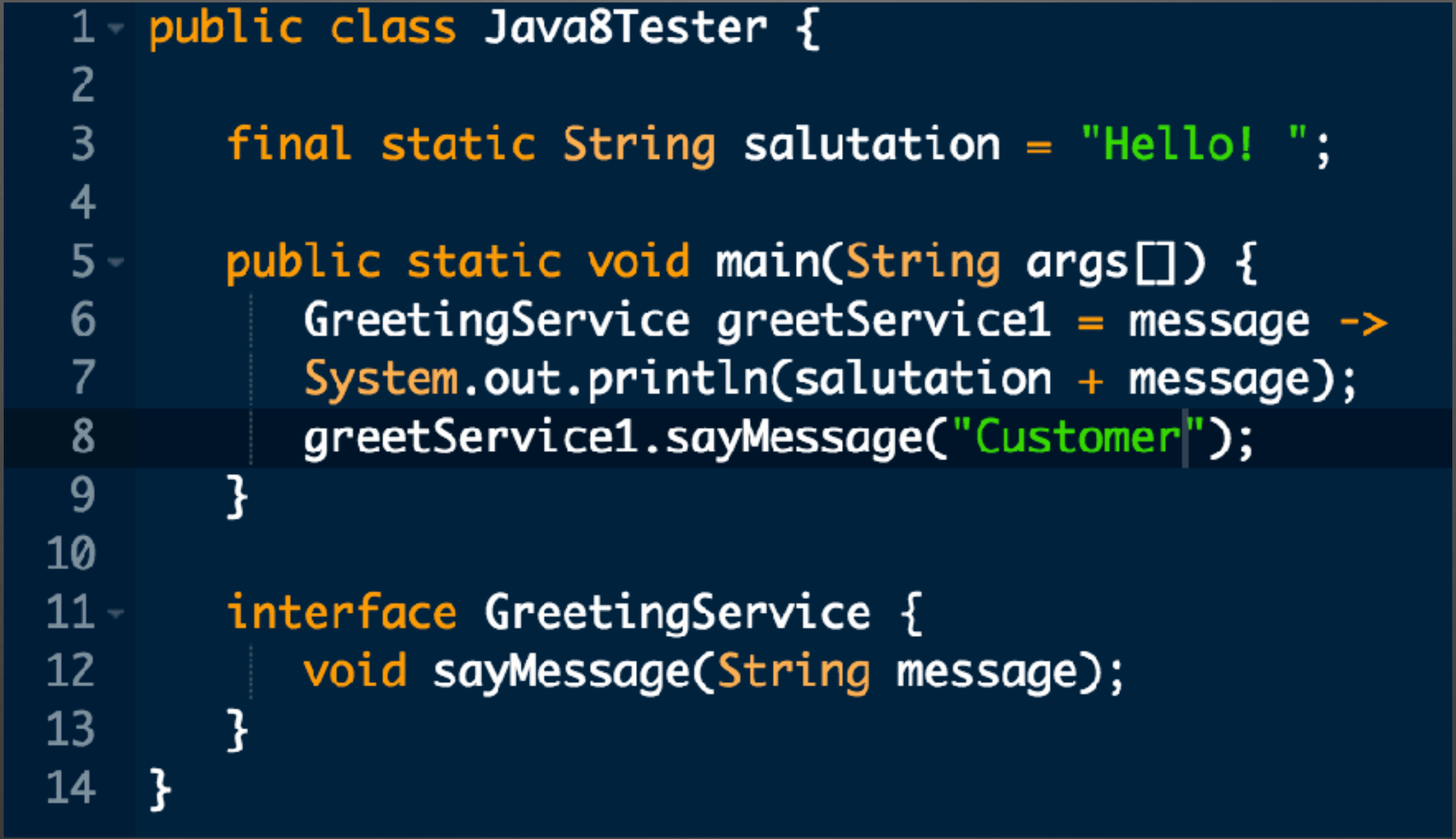


```java
public class Java8Tester {

   public static void main(String args[]) {
      Java8Tester tester = new Java8Tester();

      //with type declaration
      MathOperation addition = (int a, int b) -> a + b;

      //with out type declaration
      MathOperation subtraction = (a, b) -> a - b;

      //with return statement along with curly braces
      MathOperation multiplication = (int a, int b) -> { return a * b; };

      //without return statement and without curly braces
      MathOperation division = (int a, int b) -> a / b;

      System.out.println("10 + 5 = " + tester.operate(10, 5, addition));
      System.out.println("10 - 5 = " + tester.operate(10, 5, subtraction));
      System.out.println("10 x 5 = " + tester.operate(10, 5, multiplication));
      System.out.println("10 / 5 = " + tester.operate(10, 5, division));

   }

   interface MathOperation {
      int operation(int a, int b);
   }

   private int operate(int a, int b, MathOperation mathOperation) {
      return mathOperation.operation(a, b);
   }
}
```

Output:

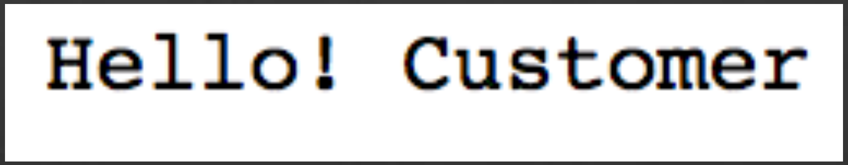


```
10 + 5 = 15
10 - 5 = 5
10 x 5 = 50
10 / 5 = 2
```



```java
public class Java8Tester {

    final static String salutation = "Hello! ";

    public static void main(String args[]) {
        GreetingService greetService1 = message ->
        System.out.println(salutation + message);
        greetService1.sayMessage("Customer");
    }

    interface GreetingService {
        void sayMessage(String message);
    }
}
```

Output:



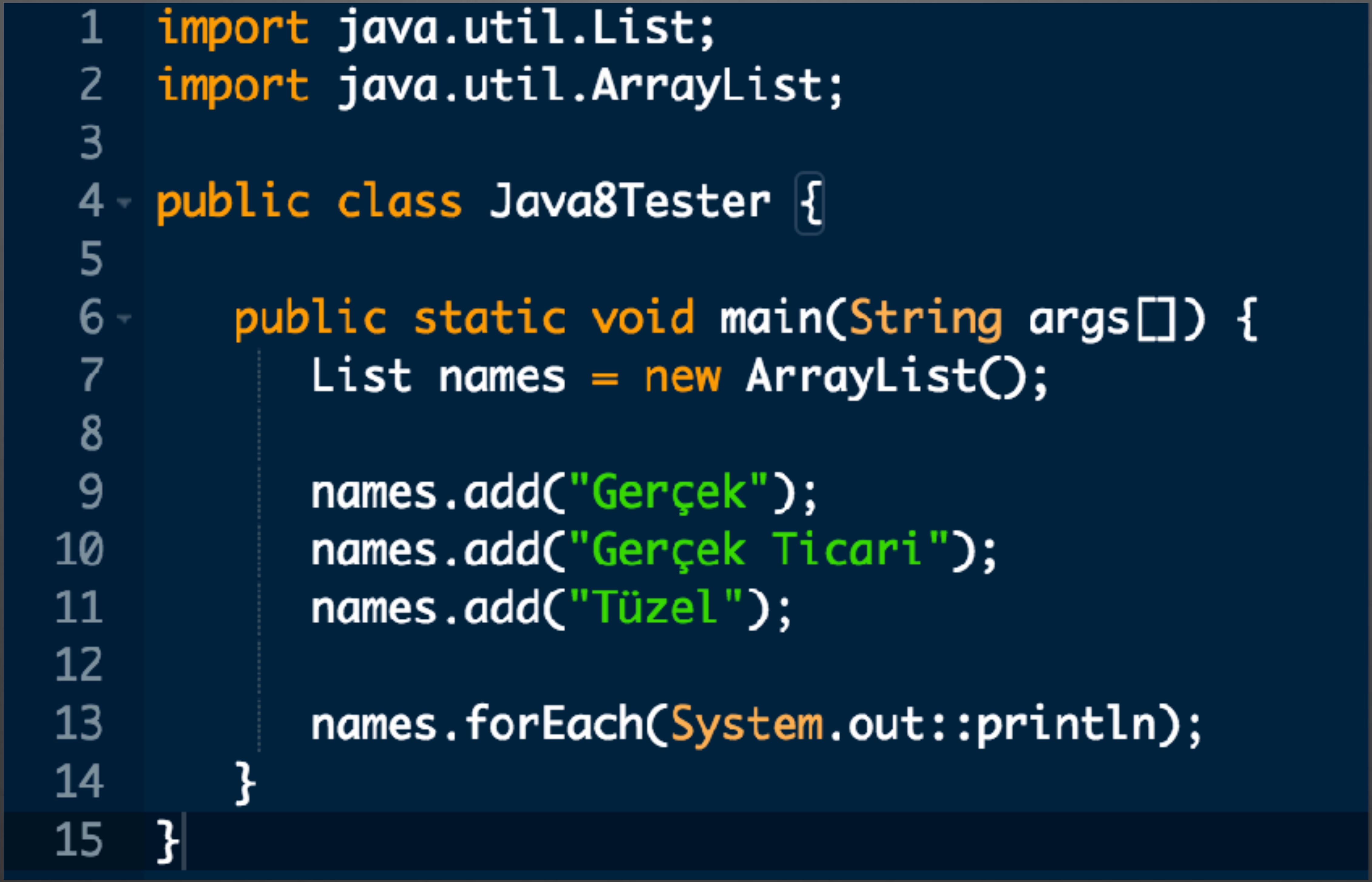
```
Hello! Customer
```

Method References

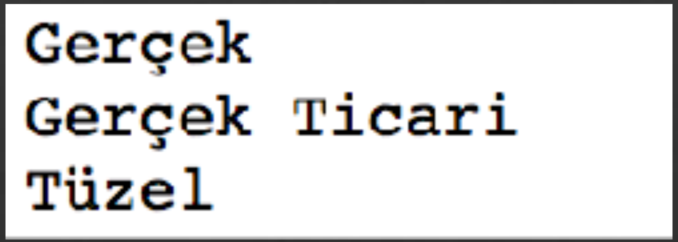
Method references help to point to methods by their names. A method reference is described using "::" symbol. A method reference can be used to point the following types of methods –

- Static methods
- Instance methods
- Constructors using new operator (TreeSet::new)

go to the next page to see examples


```java
import java.util.List;
import java.util.ArrayList;

public class Java8Tester {

    public static void main(String args[]) {
        List names = new ArrayList();

        names.add("Gerçek");
        names.add("Gerçek Ticari");
        names.add("Tüzel");

        names.forEach(System.out::println);
    }
}
```

Output:

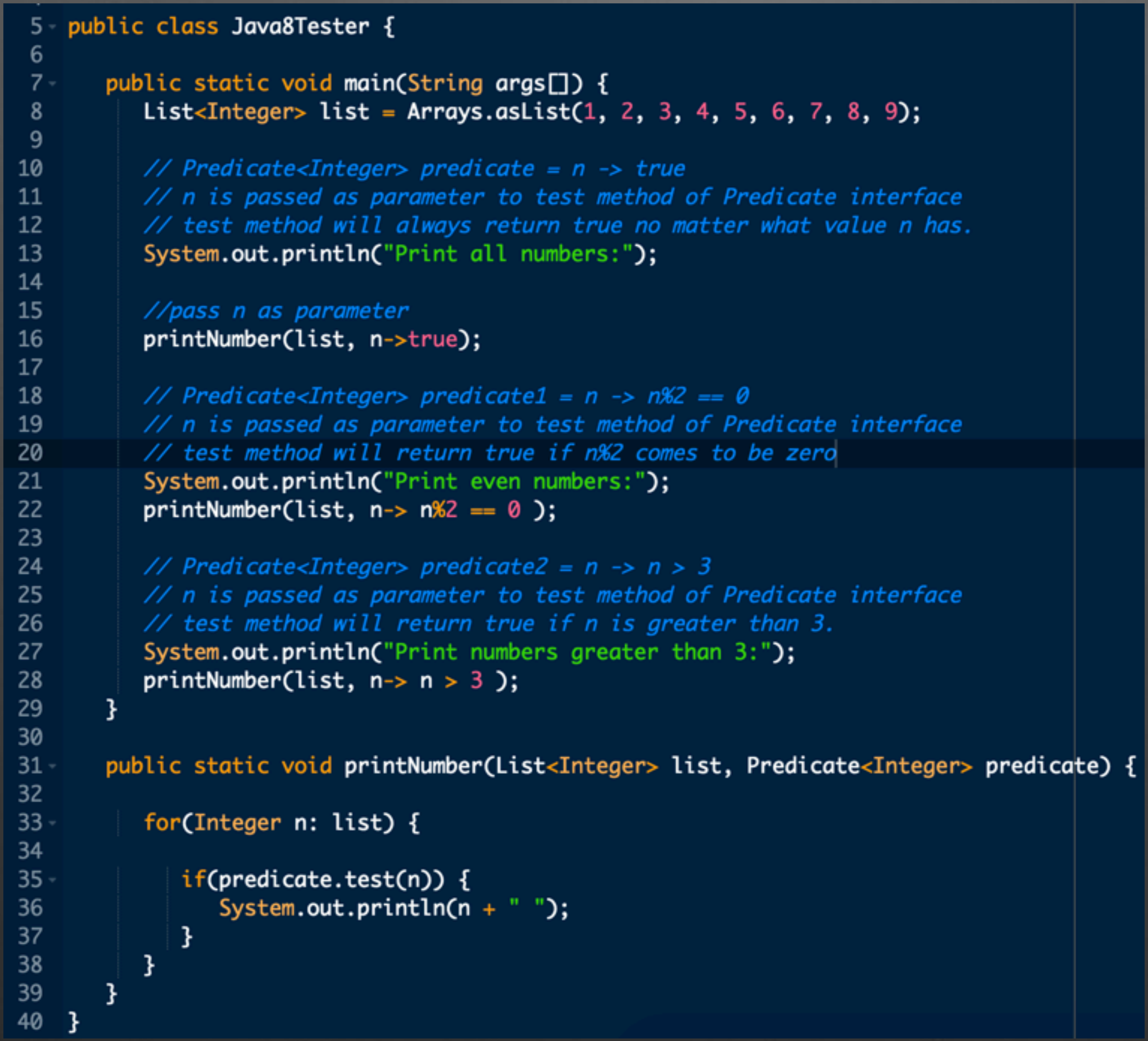

```
Gerçek
Gerçek Ticari
Tüzel
```

Functional Interfaces

Interfaces with a single abstract method are called functional interfaces.

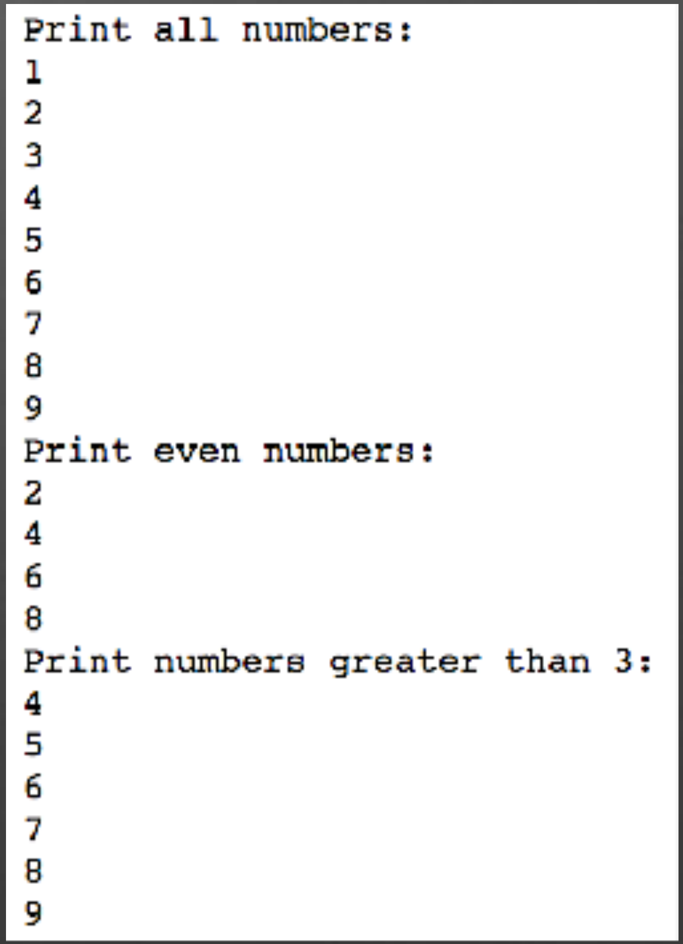
For example, a Comparable interface with a single method ‘compareTo’ is used for comparison purpose. Java 8 has defined a lot of functional interfaces to be used extensively in lambda expressions.

go to the next page to see examples



```java
public class Java8Tester {

   public static void main(String args[]) {
      List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

      // Predicate<Integer> predicate = n -> true
      // n is passed as parameter to test method of Predicate interface
      // test method will always return true no matter what value n has.
      System.out.println("Print all numbers:");

      //pass n as parameter
      printNumber(list, n->true);

      // Predicate<Integer> predicate1 = n -> n%2 == 0
      // n is passed as parameter to test method of Predicate interface
      // test method will return true if n%2 comes to be zero
      System.out.println("Print even numbers:");
      printNumber(list, n-> n%2 == 0 );

      // Predicate<Integer> predicate2 = n -> n > 3
      // n is passed as parameter to test method of Predicate interface
      // test method will return true if n is greater than 3.
      System.out.println("Print numbers greater than 3:");
      printNumber(list, n-> n > 3 );
   }

   public static void printNumber(List<Integer> list, Predicate<Integer> predicate) {

      for(Integer n: list) {

         if(predicate.test(n)) {
            System.out.println(n + " ");
         }
      }
   }
}
```

Output:

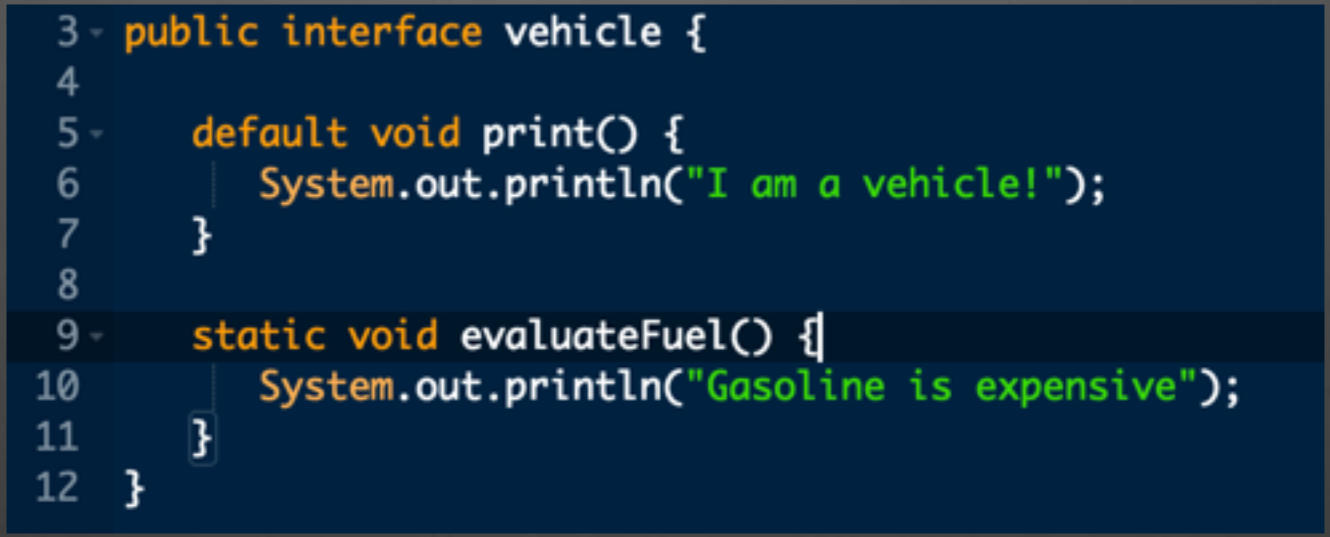


```
Print all numbers:
1
2
3
4
5
6
7
8
9
Print even numbers:
2
4
6
8
Print numbers greater than 3:
4
5
6
7
8
9
```

Default Methods

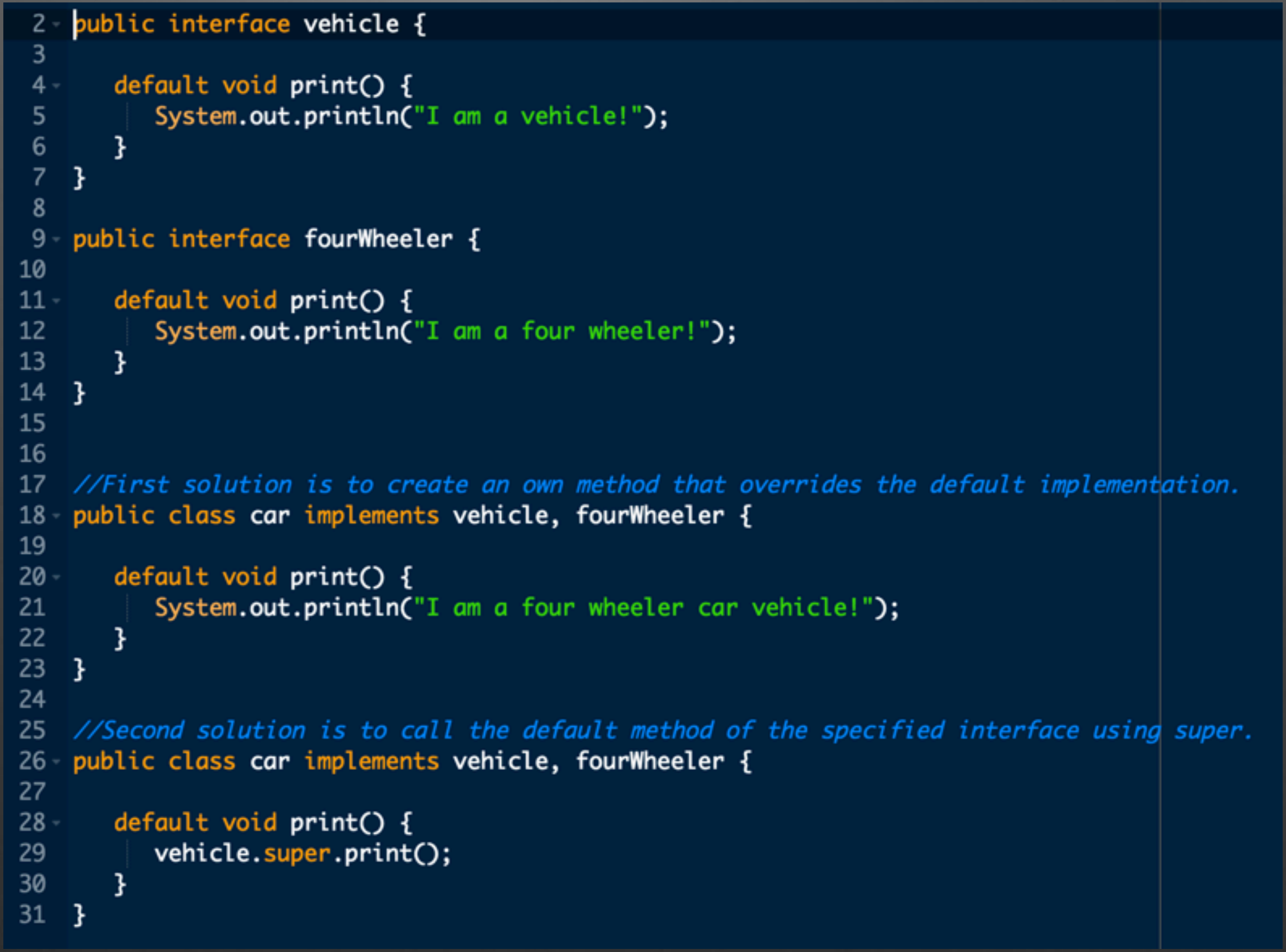
Java 8 introduces a new concept of default method implementation in interfaces.

Syntax:

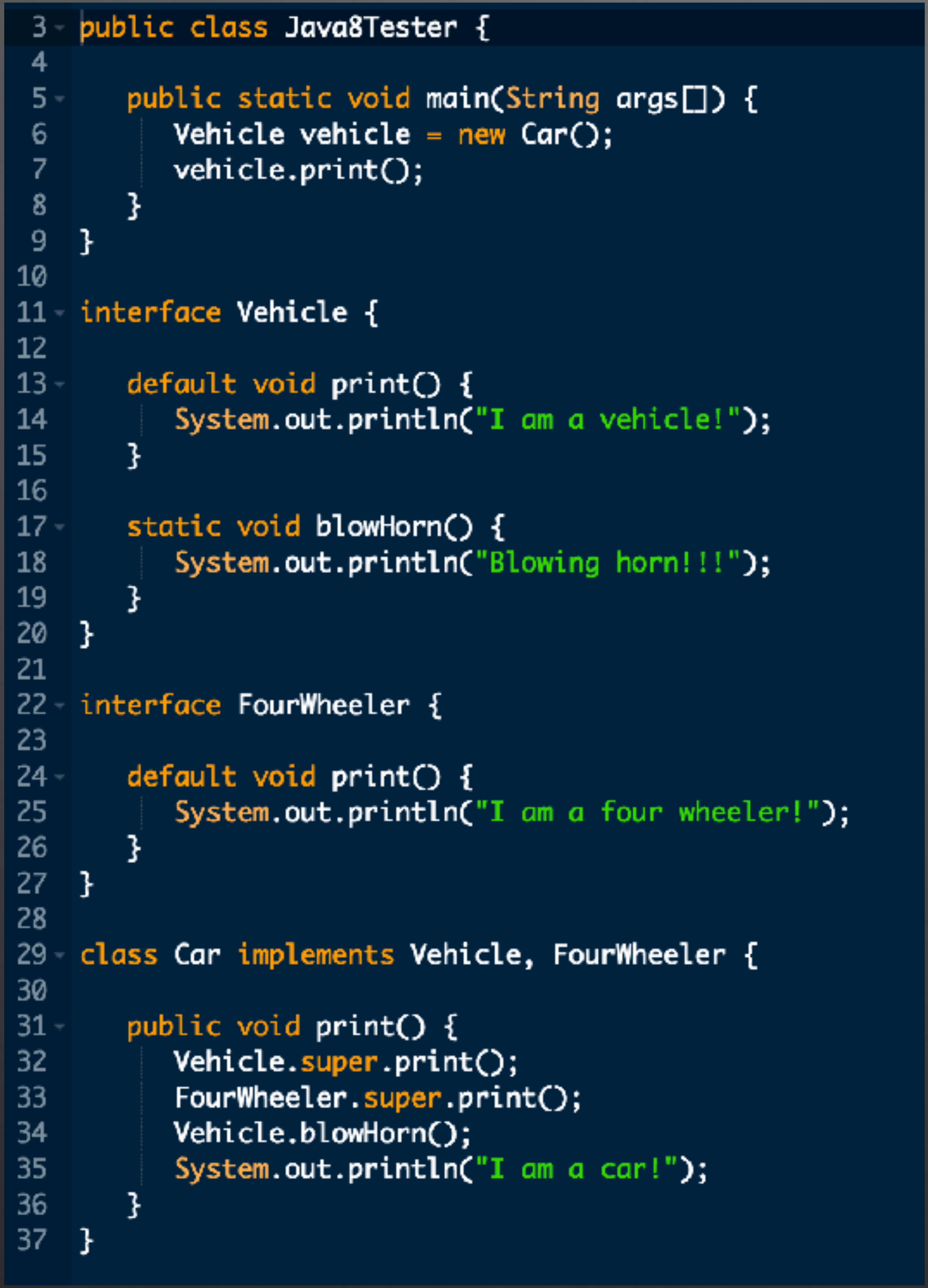


```java
public interface vehicle {

    default void print() {
        System.out.println("I am a vehicle!");
    }

    static void evaluateFuel() {
        System.out.println("Gasoline is expensive");
    }
}
```

With default functions in interfaces, there is a possibility that a class is implementing two interfaces with same default methods. The following code explains how this ambiguity can be resolved.

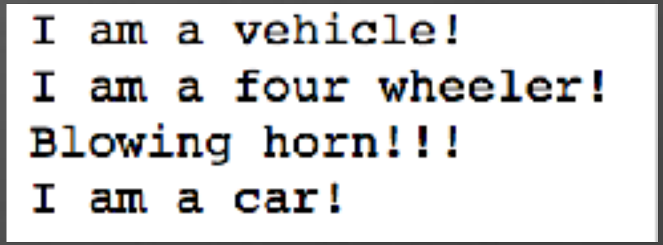


```java
public interface vehicle {

    default void print() {
        System.out.println("I am a vehicle!");
    }
}

public interface fourWheeler {

    default void print() {
        System.out.println("I am a four wheeler!");
    }
}


//First solution is to create an own method that overrides the default implementation.
public class car implements vehicle, fourWheeler {

    default void print() {
        System.out.println("I am a four wheeler car vehicle!");
    }
}

//Second solution is to call the default method of the specified interface using super.
public class car implements vehicle, fourWheeler {

    default void print() {
        vehicle.super.print();
    }
}
```



```java
public class Java8Tester {

    public static void main(String args[]) {
        Vehicle vehicle = new Car();
        vehicle.print();
    }
}

interface Vehicle {

    default void print() {
        System.out.println("I am a vehicle!");
    }

    static void blowHorn() {
        System.out.println("Blowing horn!!!");
    }
}

interface FourWheeler {

    default void print() {
        System.out.println("I am a four wheeler!");
    }
}

class Car implements Vehicle, FourWheeler {

    public void print() {
        Vehicle.super.print();
        FourWheeler.super.print();
        Vehicle.blowHorn();
        System.out.println("I am a car!");
    }
}
```

Output:



```
I am a vehicle!
I am a four wheeler!
Blowing horn!!!
I am a car!
```

Java 8 Date-Time Api

- With the new Date-Time API, improvements come about in terms of ease of use, clarity and thread-safety.
- The components (class, interface, enum, etc.) of this new API are in the java.time package.
- In general, the private constructor methods of the classes in the java.time package, so they can not be created with the new key expression.
- Instead, new objects can be created with methods such as now, of, from, and parse.

- The classes in the java.time package are immutable.
- For this reason, once an object has been created, the data in it can not be edited. This makes existing classes thread-safe.
- In this presentation, we will examine the use of the LocalDate, LocalTime, LocalDateTime, ZoneId, and ZonedDateTime classes.

Java 8 Optional Class

- A Java developer's nightmare dream is to deal with exceptions to NullPointerException.
- Added Optional class in Java 8
- Optional objects have been created to easily check null fields.
- An Optional object is created by the various static methods of the Optional class. These are empty, of, and ofNullable.

Optional Class Static Methods

- **empty**—Creates a new Optional object.
- **of**—Wraps an object with Optinal. It does not accept a null value as a parameter.
- **ofNullable** — Wraps an object with Optinal. Accepts a null value as a parameter.

Java Stream API

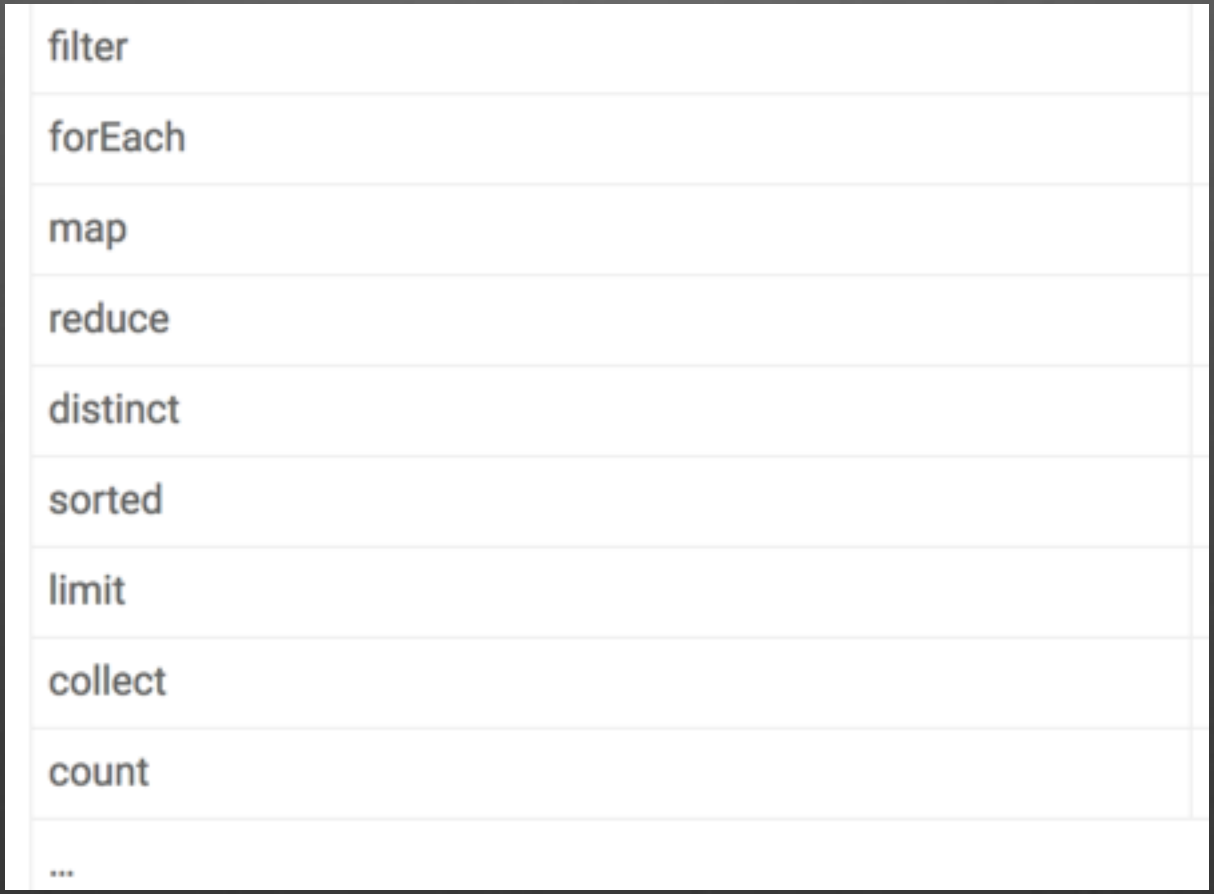
Stream API innovation has been introduced in Java 8 to make it easier to manipulate bulk data.

The Stream API is working on bulk data. The first thing to come to mind about bulk data is undoubtedly the arrays (such as byte [], String []) and Java Collection API components (such as List, Set)

The Stream API allows easy and efficient use of various common operations on such bulk data.

The most common ones used in these operations are as follows.

These operations and more are found in the java.util.stream.Stream interface.



filter
forEach
map
reduce
distinct
sorted
limit
collect
count
...

Parallel Stream

if it does not need to be executed sequential from the methods in the Stream interface, it can run in parallel on the CPU.

In this way, it is possible to use CPU cores in full efficiency.
It is quite easy to get a parallel stream in the Stream API.

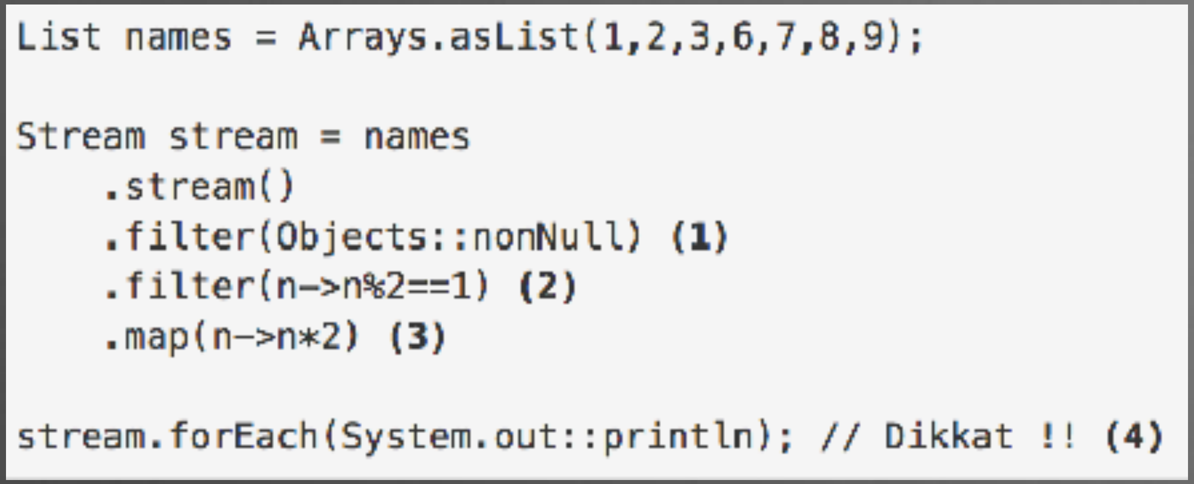
```
List ints = Arrays.asList(1, 3, 5, 7, 9, 11, 13, 15);
Stream stream = ints.stream(); // Sequential
Stream parallelStream = stream.parallel(); // Parallel
```

Lazy & Eager operations

In the literature, a Lazy process is to be made as late, delayed, whereas Eager is used to represent that the process to be done is to be executed.

Some operations in Stream API are Lazy and others are run as Eager. Chained tasks with lazy behavior do not run until an Eager operation.

go to the next page to see examples



```
List names = Arrays.asList(1,2,3,6,7,8,9);

Stream stream = names
    .stream()
    .filter(Objects::nonNull) (1)
    .filter(n->n%2==1) (2)
    .map(n->n*2) (3)

stream.forEach(System.out::println); // Dikkat !! (4)
```

For example, 2 filters and 3 map operations are lazy operations.
When the code fragment is run in this way, neither a filtering nor a conversion will be done.
Here, only the Stream object is prepared.
Lazy operations are not processed unless necessary.

However, if this Stream object is encountered with Eager operation, Lazy operations in previous chains will work immediately.

That is, the operation in (4) is the trigger of operations in (1) (2) (3).

Nashorn JavaScript

- Nashorn is a JS engine developed for java 8 by Oracle
- Allows to use the JavaScript language in the JVM environment
- At this point, developers will be able to add Javascript combinations into Java code. At the same time, the command jjs is also available to run Nashorn from the command line.