**Lambda Expressions Syntax**

* [**Email Article**](https://www.developer.com/e-mail/java/start-using-java-lambda-expressions.html)
* [**Print Article**](https://www.developer.com/print/java/start-using-java-lambda-expressions.html)

The basic syntax of a lambda is either:

(*parameters*) ->*expression*or         (parameters) ->{ statements; }

The following are examples of Java lambda expressions:

*1.*(*) -> 5* // takes no value and returns 5

2.  x -> 2 \* x            // takes a number and returns the result of doubling it

*3.*(*x, y) -> x – y*  // takes two numbers and returns their difference

4.  (int x, int y) -> x + y       // takes two integers and returns their sum

5.  (*String s) -> System.out.print(s)* // takes a string and prints it to console without returning anything

**Basic Java Lambdas Examples**

Now that we have an idea of what lambdas are, let us start with some basic examples. In this section, we will see how lambda expressions affect the way we code. Having a list of players, the “for loop”, as programmers often refers to the for statement, can be translated in Java SE 8 as below:

String[] atp = {"Rafael Nadal", "Novak Djokovic", "Stanislas Wawrinka", "David Ferrer", "Roger Federer", "Andy Murray", "Tomas Berdych", "Juan Martin Del Potro"};

List<String> players = Arrays.asList(atp);

// Old looping

for (String player : players) {

System.out.print(player + "; ");

}

// Using lambda expression and functional operations

players.forEach((player) -> System.out.print(player + "; "));

// Using double colon operator in Java 8

players.forEach(System.out::println);

As you can see, lambda expressions can reduced our code to one single line. Another example is in a graphical user interface application, when anonymous classes can be replaced with lambda expressions. The same thing happens when implementing the Runnable interface:

// Using anonymous innerclass

btn.setOnAction(new EventHandler<ActionEvent>() {

              @Override

public void handle(ActionEvent event) {

System.out.println("Hello World!");

            }

        });

// Using lambda expression

btn.setOnAction(event -> System.out.println("Hello World!"));

Here is how we can write a Runnable using lambdas:

// Using anonymous innerclass

new Thread(new Runnable() {

@Override

public void run() {

System.out.println("Hello world !");

}

}).start();

// Using lambda expression

new Thread(() -> System.out.println("Hello world !")).start();

// Using anonymous innerclass

Runnable race1 = new Runnable() {

@Override

public void run() {

System.out.println("Hello world !");

}

};

// Using lambda expression

Runnable race2 = () -> System.out.println("Hello world !");

// Run em!

race1.run();

race2.run();

The Runnable lambda expression, which uses the block format, converts five lines of code into one statement. Going further, in the next section we will use lambdas for sorting collections.

**Sorting Collections with Java Lambdas**

In Java, the Comparator class is used for sorting collections. In the following examples, we will sort a list of players based on name, surname, name length and last name letter. We will first sort them as we did before, using anonymous inner classes, and then reduce our code using lambda expressions.

In the first example, we will sort our list by name. Using the old way, this looks like this:

String[] players = {"Rafael Nadal", "Novak Djokovic", "Stanislas Wawrinka", "David Ferrer", "Roger Federer", "Andy Murray", "Tomas Berdych", "Juan Martin Del Potro", "Richard Gasquet", "John Isner"};

// Sort players by name using anonymous innerclass

Arrays.sort(players, new Comparator<String>() {

@Override

public int compare(String s1, String s2) {

return (s1.compareTo(s2));

}

});

With lambdas, the same thing can be achieved like this:

// Sort players by name using lambda expression

Comparator<String> sortByName = (String s1, String s2) -> (s1.compareTo(s2));

Arrays.sort(players, sortByName);

// or this

Arrays.sort(players, (String s1, String s2) -> (s1.compareTo(s2)));

The remaining sorts are listed below. The same as the above example, the code applies a Comparator by using an anonymous inner class and a couple of lambda expressions:

// Sort players by surname using anonymous innerclass

Arrays.sort(players, new Comparator<String>() {

@Override

public int compare(String s1, String s2) {

return (s1.substring(s1.indexOf(" ")).compareTo(s2.substring(s2.indexOf(" "))));

}

});

// Sort players by surname using lambda expression

Comparator<String> sortBySurname = (String s1, String s2) -> (s1.substring(s1.indexOf(" ")).compareTo(s2.substring(s2.indexOf(" "))));

Arrays.sort(players, sortBySurname);

// or this

Arrays.sort(players, (String s1, String s2) -> (s1.substring(s1.indexOf(" ")).compareTo(s2.substring(s2.indexOf(" ")))));

// Sort players by name lenght using anonymous innerclass

Arrays.sort(players, new Comparator<String>() {

@Override

public int compare(String s1, String s2) {

return (s1.length() - s2.length());

}

});

// Sort players by name lenght using lambda expression

Comparator<String> sortByNameLenght = (String s1, String s2) -> (s1.length() - s2.length());

Arrays.sort(players, sortByNameLenght);

// or this

Arrays.sort(players, (String s1, String s2) -> (s1.length() - s2.length()));

// Sort players by last letter using anonymous innerclass

Arrays.sort(players, new Comparator<String>() {

@Override

public int compare(String s1, String s2) {

return (s1.charAt(s1.length() - 1) - s2.charAt(s2.length() - 1));

}

});

// Sort players by last letter using lambda expression

Comparator<String> sortByLastLetter = (String s1, String s2) -> (s1.charAt(s1.length() - 1) - s2.charAt(s2.length() - 1));

Arrays.sort(players, sortByLastLetter);

// or this

Arrays.sort(players, (String s1, String s2) -> (s1.charAt(s1.length() - 1) - s2.charAt(s2.length() - 1)));

That’s all, pretty straightforward. We will explore more lambdas capabilities in the next section, where we will combine them with streams.

**Working with Java Lambdas and Streams**

            Streams are wrappers around collections that use lambdas pervasively. They support many operations that use lambdas, like map, filter, limit, sorted, count, min, max, sum, collect and others. Also, streams use lazy evaluation and they are not actually reading all the data and methods like getFirst() can end the stream. In the next examples, we will explore what lambdas and streams can do. We created a Person class and use this class to add some data to a list that will be used in further streams operations. The Person class is just a simple POJO:

public class Person {

private String firstName, lastName, job, gender;

private int salary, age;

public Person(String firstName, String lastName, String job, String gender, int age, int salary)       {

this.firstName = firstName;

this.lastName = lastName;

this.gender = gender;

this.age = age;

this.job = job;

this.salary = salary;

}

// Getter and Setter

. . . . .

}

Going further, we will create two lists, both containing Person objects:

            List<Person> javaProgrammers = new ArrayList<Person>() {

            {

                add(new Person("Elsdon", "Jaycob", "Java programmer", "male", 43, 2000));

                add(new Person("Tamsen", "Brittany", "Java programmer", "female", 23, 1500));

                add(new Person("Floyd", "Donny", "Java programmer", "male", 33, 1800));

                add(new Person("Sindy", "Jonie", "Java programmer", "female", 32, 1600));

                add(new Person("Vere", "Hervey", "Java programmer", "male", 22, 1200));

                add(new Person("Maude", "Jaimie", "Java programmer", "female", 27, 1900));

                add(new Person("Shawn", "Randall", "Java programmer", "male", 30, 2300));

                add(new Person("Jayden", "Corrina", "Java programmer", "female", 35, 1700));

                add(new Person("Palmer", "Dene", "Java programmer", "male", 33, 2000));

                add(new Person("Addison", "Pam", "Java programmer", "female", 34, 1300));

            }

        };

        List<Person> phpProgrammers = new ArrayList<Person>() {

            {

                add(new Person("Jarrod", "Pace", "PHP programmer", "male", 34, 1550));

                add(new Person("Clarette", "Cicely", "PHP programmer", "female", 23, 1200));

                add(new Person("Victor", "Channing", "PHP programmer", "male", 32, 1600));

                add(new Person("Tori", "Sheryl", "PHP programmer", "female", 21, 1000));

                add(new Person("Osborne", "Shad", "PHP programmer", "male", 32, 1100));

                add(new Person("Rosalind", "Layla", "PHP programmer", "female", 25, 1300));

                add(new Person("Fraser", "Hewie", "PHP programmer", "male", 36, 1100));

                add(new Person("Quinn", "Tamara", "PHP programmer", "female", 21, 1000));

                add(new Person("Alvin", "Lance", "PHP programmer", "male", 38, 1600));

                add(new Person("Evonne", "Shari", "PHP programmer", "female", 40, 1800));

            }

        };

Let’s now use the forEach method to iterate the above lists:

System.out.println("Show programmers names:");

javaProgrammers.forEach((p) -> System.out.printf("%s %s; ", p.getFirstName(), p.getLastName()));

phpProgrammers.forEach((p) -> System.out.printf("%s %s; ", p.getFirstName(), p.getLastName()));

We will now use the same forEach method and try to increase programmer’s salary by 5%:

System.out.println("Increase salary by 5% to programmers:");

Consumer<Person> giveRaise = e -> e.setSalary(e.getSalary() / 100 \* 5 + e.getSalary());

javaProgrammers.forEach(giveRaise);

phpProgrammers.forEach(giveRaise);

Another useful method is the filter method. To make use of that, let’s show PHP programmers that earn more than $1,400:

System.out.println("Show PHP programmers that earn more than $1,400:")

phpProgrammers.stream()

.filter((p) -> (p.getSalary() > 1400))

.forEach((p) -> System.out.printf("%s %s; ", p.getFirstName(), p.getLastName()));

We can also define filters and then reuse them in further operations:

// Define some filters

Predicate<Person> ageFilter = (p) -> (p.getAge() > 25);

Predicate<Person> salaryFilter = (p) -> (p.getSalary() > 1400);

Predicate<Person> genderFilter = (p) -> ("female".equals(p.getGender()));

System.out.println("Show female PHP programmers that earn more than $1,400 and are older than 24 years:");

phpProgrammers.stream()

.filter(ageFilter)

.filter(salaryFilter)

.filter(genderFilter)

.forEach((p) -> System.out.printf("%s %s; ", p.getFirstName(), p.getLastName()));

// Reuse filters

System.out.println("Show female Java programmers older than 24 years:");

javaProgrammers.stream()

.filter(ageFilter)

.filter(genderFilter)

.forEach((p) -> System.out.printf("%s %s; ", p.getFirstName(), p.getLastName()));

Results can also be limited, if we use the limit method:

System.out.println("Show first 3 Java programmers:");

javaProgrammers.stream()

.limit(3)

.forEach((p) -> System.out.printf("%s %s; ", p.getFirstName(), p.getLastName()));

System.out.println("Show first 3 female Java programmers:");

javaProgrammers.stream()

.filter(genderFilter)

.limit(3)

.forEach((p) -> System.out.printf("%s %s; ", p.getFirstName(), p.getLastName()));

What about sorting? Can we do that with streams? The answer is yes, we can. In the examples below, we will sort Java programmers by name and salary, collect them into a list and then show the list:

System.out.println("Sort and show the first 5 Java programmers by name:");

List<Person> sortedJavaProgrammers = javaProgrammers

.stream()

.sorted((p, p2) -> (p.getFirstName().compareTo(p2.getFirstName())))

.limit(5)

.collect(toList());

sortedJavaProgrammers.forEach((p) -> System.out.printf("%s %s; %n", p.getFirstName(), p.getLastName()));

System.out.println("Sort and show Java programmers by salary:");

sortedJavaProgrammers = javaProgrammers

.stream()

.sorted((p, p2) -> (p.getSalary() - p2.getSalary()))

.collect(toList());

sortedJavaProgrammers.forEach((p) -> System.out.printf("%s %s; %n", p.getFirstName(), p.getLastName()));

If we are interested only in the lowest and the highest salary, faster than sorting and choosing the first (or the last) are the min and max methods:

System.out.println("Get the lowest Java programmer salary:");

Person pers = javaProgrammers

.stream()

.min((p1, p2) -> (p1.getSalary() - p2.getSalary()))

.get()

System.out.printf("Name: %s %s; Salary: $%,d.", pers.getFirstName(), pers.getLastName(), pers.getSalary())

System.out.println("Get the highest Java programmer salary:");

Person person = javaProgrammers

.stream()

.max((p, p2) -> (p.getSalary() - p2.getSalary()))

.get()

System.out.printf("Name: %s %s; Salary: $%,d.", person.getFirstName(), person.getLastName(), person.getSalary())

We have already seen above an example on how the collect method works. Combined with the map method, we can use collect method to gather our results in a String, in a Set or in a TreeSet:

System.out.println("Get PHP programmers first name to String:");

String phpDevelopers = phpProgrammers

.stream()

.map(Person::getFirstName)

.collect(joining(" ; "));    // this can be used as a token in further operations

System.out.println("Get Java programmers first name to Set:");

Set<String> javaDevFirstName = javaProgrammers

.stream()

.map(Person::getFirstName)

.collect(toSet());

System.out.println("Get Java programmers last name to TreeSet:");

TreeSet<String> javaDevLastName = javaProgrammers

.stream()

.map(Person::getLastName)

.collect(toCollection(TreeSet::new));

Streams can also be parallel. An example is below:

System.out.println("Calculate total money spent for paying Java programmers:");

int totalSalary = javaProgrammers

.parallelStream()

.mapToInt(p -> p.getSalary())

.sum();

To obtain various summary data about the elements of a stream we can use the summaryStatistics method. Going further, we then have access to methods like getMax, getMin, getSum or getAverage:

//Get count, min, max, sum, and average for numbers

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

IntSummaryStatistics stats = numbers

.stream()

.mapToInt((x) -> x)

.summaryStatistics();

System.out.println("Highest number in List : " + stats.getMax());

System.out.println("Lowest number in List : " + stats.getMin());

System.out.println("Sum of all numbers : " + stats.getSum());

System.out.println("Average of all numbers : " + stats.getAverage());