

Performance of Lambda Expressions in High Level Programming Languages

Todor Todorov^{1,2}, Nikolay Noev²

¹ St. Cyril and St. Methodius University of Veliko Tarnovo, 3G. Kozarev Str.,
5000 Veliko Turnovo, Bulgaria

² Institute of Mathematics and Informatics - Bulgarian Academy of Sciences, 8,
G. Bonchev Str., 1113 Sofia, Bulgaria

Abstract – Functional programming is a programming paradigm that is becoming increasingly popular among software developers. This is due in part to the rise of distributed systems and the need for more robust and scalable code. In the paper is presented an overview of the syntax and capabilities of Lambda expressions in three programming languages – C#, Java and Python. Performance of programming language constructions is an important research task. Some popular topics for investigation are comparison of programming languages efficiency in fields like bioinformatics or classification of Lambda expression usage and their productiveness. In the current study the performance of Lambda expressions is tested with three specific test cases and the results are compared to alternative technologies that could be used to solve similar problems. The results shows that speed performance of C# is the best from the compared languages and that List Comprehensions is the optimal method for collection filtering in Python.

Keywords – lambda expressions, programming languages, comparison of performance.

DOI: 10.18421/TEM124-34

<https://doi.org/10.18421/TEM124-34>

Corresponding author: Todor Todorov,
St. Cyril and St. Methodius University of Veliko Tarnovo,
3G. Kozarev Str., 5000 Veliko Turnovo, Bulgaria


Email: t.todorov@ts.uni-vt.bg

Received: 20 June 2023.

Revised: 25 September 2023.

Accepted: 11 October 2023.

Published: 27 November 2023.

 © 2023 Todor Todorov & Nikolay Noev; published by UIKTEN. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDeriv 4.0 License.

The article is published with Open Access at <https://www.temjournal.com/>

1. Introduction

Functional programming is a key part of contemporary programming languages. It is based on mathematical functions, conditional expressions and recursion. One of the important features of functional programming is the use of lambda expressions, which gives a concise and flexible alternative to functions [1].

Lambda expressions are a relatively recent addition to the common programming languages and have the ability to simplify and streamline the way in which functions are written and used. They are anonymous functions that are used as other functions arguments or variable values. Since their introduction they have been widely adopted by developers due to the increased flexibility and concise syntax they provide.

Comparison of performance of programming concepts in different programming languages is an important research topic [2], [3], [4], [5].

In the paper are explored the concept of lambda expressions and are discussed their uses, benefits, and limitations. Also, is examined the use of lambda expressions in three popular programming languages: Java, C# and Python.

2. The Concept of Lambda Expressions

In the section is made an overview of the basic syntax and programming constructions related to Lambda expressions in JAVA, C# and Python. Examples from the three languages are presented and remarks of similarities and differences are summarized.

2.1. Java

Java 8 introduced support for lambda expressions, which allow developers to write functions as concise, anonymous functions that can be passed as arguments to other functions or stored in variables [6], [7], [8], [9].

Lambda expressions in Java are written using the syntax:

```
(arguments) -> expression
```

where the arguments are a comma-separated list of parameters, and the expression is the body of the function.

Lambda expressions can be used in a variety of contexts in Java, including as arguments to functional interfaces, as the implementation of anonymous inner classes, and as the implementation of functional methods. For example, as parameters to functions like the **forEach** method, which can be used to process elements of a stream in a functional manner [10]:

```
Stream<Integer> stream = Arrays.asList(1, 2, 3, 4, 5).stream();  
stream.forEach(n -> System.out.println(n));
```

A list of integers is presented and the **forEach** method outputs each element. A lambda expression is used as a parameter of the **forEach** method.

Functional Interfaces

Functional interfaces are interfaces that declare exactly one abstract method. In Java, functional interfaces can be used as a source for lambda expression. Next is presented a functional interface to sort a list of strings in ascending order:

```
List<String> list = Arrays.asList("dog", "cat", "bird");  
list.sort((a, b) -> a.compareTo(b));  
System.out.println(list);
```

Anonymous Inner Classes

Lambda expressions can also be used as a concise alternative to a class that is defined and instantiated in one line of code, without a name. Let's consider the following anonymous inner class:

```
new Thread(new Runnable() {  
public void run() {  
System.out.println("Hello from the thread");  
}  
}).start();
```

The code can be rewritten with a lambda expression, as shown in the following listing:

```
new Thread(() -> System.out.println("Hello from the  
thread")).start
```

2.2. C#

C# 3.0 introduced support for lambda expressions, which are similar in syntax and functionality to those in Java [11], [12].

Lambda expressions in C# are written using the syntax:

```
(arguments) => expression.
```

Lambda expressions can be used in a variety of contexts in C#, including as arguments to delegates, as the implementation of anonymous methods, and as the implementation of LINQ query expressions:

Delegates

Delegates are a type-safe, object-oriented mechanism for calling methods, and are widely used in C# as event handlers and callbacks. Lambda expressions can be used as arguments to delegates to provide a flexible way to define and use functions.

In the next passage is shown a delegate type **Func** that takes two integer arguments and returns an integer. A lambda expression is used as a delegate variable value:

```
delegate int Func(int a, int b);  
  
Func sum = (a, b) => a + b;
```

Anonymous Methods

Lambda expressions can also be used as a good alternative to anonymous methods. An anonymous method is a method that is defined and instantiated in one line of code, without a name. Let's create a thread using an anonymous method:

```
new Thread(delegate() { Console.WriteLine("Hello from the  
thread"); }).Start();
```

This code can be rewritten using a lambda expression:

```
new Thread(() => Console.WriteLine("Hello from the  
thread")).Start();
```

LINQ (Language Integrated Query)

LINQ query expressions are a core part of LINQ, and they allow developers to write queries using the syntax of the C# language.

In the next excerpt is shown a query to a list of integers to find all even numbers:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6 };  
  
IEnumerable<int> evenNumbers =  
  
    from number in numbers where number % 2 == 0  
  
    select number;  
  
    foreach (int evenNumber in evenNumbers)  
  
        { Console.WriteLine(evenNumber); }
```

2.3. Python

Python is a dynamically typed, interpreted programming language that has been widely used since its initial release in 1991. Python has had support for lambda expressions since its earliest versions, which are written using the syntax lambda arguments: expression [13], [14], [15], [16].

The syntax is:

```
lambda arguments: expression
```

The arguments are the inputs to the lambda function and the expression is the result of the function. The expression can be any valid Python expression that returns a value, and it can refer to the arguments in the same way as in a regular function definition.

Lambda expressions in Python are often used as anonymous functions that are passed as arguments to other functions. In the following listings a lambda expression is used as the key function when sorting a list of tuples:

```
list = [(1, 2), (3, 1), (2, 3)]
sorted_list = sorted(list, key=lambda x: x[1])
```

One of the main use cases for lambda expressions in Python is as a compact way to specify a function for a higher-order function. One such function is the map function in Python. It transforms a collection to another collection by applying a function to all elements:

```
numbers = [1, 2, 3, 4, 5]
squaredNumbers = list(map(lambda x: x**2, numbers))
print(squaredNumbers)
```

The map function is applied to the list of numbers, and the lambda expression *lambda x: x**2* is used to specify the function that is applied to each number. The result is a list of the squares of the numbers.

Table 1. Lambda functions support

LANGUAGE	SYNTAX	HIGHER-ORDER FUNCTIONS	CLOSURE	TYPE INFERENCE
JAVA	(arguments) -> expression or (arguments) -> { statements; }	Yes	Yes	Yes
C#	(arguments) => expression or (arguments) => { statements; }	Yes	Yes	Yes
PYTHON	lambda arguments: expression	Yes	Yes	No

Another use case for lambda expressions in Python is as a convenient way to create small, throw-away functions for one-off operations. The next source code sorts a list of strings by the length of the strings:

```
strings = ['hello', 'world', 'foo', 'bar']
sortedStrings = sorted(strings, key=lambda s: len(s))
print(sortedStrings)
```

The sorted function is used to sort the list of strings, and the expression *lambda s: len(s)* is used to specify the key function that is used to determine the sort order. The result is a list of strings sorted by the length of the strings.

3. Comparison of Performance

In the section are presented main research results of the paper. In 3.1 are summarized Lambda functions support capabilities of different programming languages. In 3.2 is described the test environment used for experiments. This includes both hardware and software prerequisites. In 3.3 are presented all the test results about the performance of Lambda expressions in C#, JAVA and Python and filtering capabilities of Python.

3.1. Lambda Functions Support

In Table 1 are summarized results of comparison about the support of lambda functions in Java, C#, and Python:

- **Syntax:** The syntax for defining a lambda expression in each language;
- **Higher-Order Functions:** Whether the language supports higher-order functions;
- **Closure:** Whether the lambda functions in each language can capture variables from the surrounding scope;
- **Type Inference:** Whether the language supports type inference, which automatically determines the types of variables based on the context in which they are used.

As can be seen from the table, all three languages support lambda functions and higher-order functions, and all three languages support closures. However, only Java and C# support type inference.

3.2. Test Environment

The tests performed aimed to calculate the execution time for some predefined functionalities with and without the usage of lambda functions on the three considered programming languages. The test environment consists of Intel Core CPU i7-11700, 8 GB DDR4 and operating system Windows 11 Pro. The following versions of programming languages are used: C# 11.0 with .NET 7.0, Java SE 19, Python 3.11.2.

All the tests are applied on a **List** of objects from class **Employee** that contains three fields **id(int)**, **name(String)**, **salary(double)**. All the functionalities are similar to C#, Java and Python and will be demonstrated using the C# syntax. Only some specific methods will be presented for Java and Python

The class **Employee** also implements **IComparable** interface and implements **CompareTo** method so that objects from the class could be sorted according to the value of the **salary** field. Structure of the class **Employee** is presented in Figure 1.

```

class Employee: IComparable<Employee>
{
    private int id;
    private String name;
    private double salary;

    1 reference
    public Employee(int id, String name, double salary) ...
    0 references
    public int getId() ...
    0 references
    public void setId(int id) ...
    0 references
    public String getName() ...
    0 references
    public void setName(String name) ...
    3 references
    public double getSalary() ...
    0 references
    public void setSalary(double salary) ...
    0 references
    public int CompareTo(Employee obj)
    {
        if (obj == null)
            return 1;
        if (this.salary > obj.salary)
            return 1;
        else if (this.salary < obj.salary)
            return -1;
        else
            return 0;
    }
}
    
```

Figure 1. Class Employee

CompareTo method returns a negative, positive integer or zero depending on if the first value is less than the second value or vice versa.

Before presenting the tests essentials following observation should be made:

- All the tests are performed on a **List** of 200000 objects with random generated values for the field **salary**. Pseudorandom external generator is used together with additional specialty implemented algorithm for value interval adjustments.
- For execution time measuring are used methods of **Stopwatch** class in C# and its analogues in Java and Python.
- All the tests are performed one hundred times each and is taken average time of all executions for more precise conclusions.

3.3. Test Results

First test includes sort of the list of **Employee** objects according to salary filed using Lambda expressions. The following Lambda expressions are used:

```

C#
employees.Sort((o1, o2) =>
o1.getSalary().CompareTo(o2.getSalary()));
    
```

```

Java
employees.sort((o1, o2) -> Double.compare(o1.getSalary(),
o2.getSalary()));
    
```

```

Python
emp_list = sorted(emp_list, key=lambda t: t.getSalary())
    
```

Second test applies default sorting functionality with built-in Comparer and using the **CompareTo** method or its alternatives. The code is:

```

C#
employees.Sort();
    
```

```

Java
Collections.sort(employees);
    
```

```

Python
emp_list = sorted(emp_list)
    
```

For built-in sorting in Python are investigated two alternatives. The one presented on the next listing is using overrides of comparison methods of the **Employee** class:

```

def __eq__(self, other):
    return self.salary == other.salary
    
```

```

def __lt__(self, other):
    return self.salary < other.salary
    
```

The other tested alternative is with a sorting function passed as a parameter to the sorted method:

```

def softFn(elem):
    return elem.getSalary()
    
```

Third test performs research on the filtering capabilities of Lambda functions. It measures the time for filtering the list and extract only those objects that have value for **salary** field greater than 50000. These are approximately half of the elements in the list:

```
C#
empFiltLst = employees.Where(employee =>
employee.getSalary() > 50000);
```

```
Java
Stream <Employee> empStm =
employees.stream().filter(employee -> employee.getSalary() >
50000);
```

```
Python
filteredResults = filter(lambda x: x.getSalary() > 50000,
emp_list)
```

Finally, is performed a test of filtering but extended with the time-consuming operation of transformation the filtered result to list and retrieving the number of filtered elements:

```
C#
filteredResultsCnt = empFiltLst.Count();
```

```
Java
filteredResultsCnt =
empStm.collect(Collectors.toList()).size();
```

```
Python
filteredResultsCnt = len(list(filteredResults))
```

Table 2. Test results

Test	C#	Java	Python
Sort with lambda	43.32 ms	54.36 ms	68.04 ms
Sort with comparer	41.52 ms	52.84 ms	External function 67.23 ms
			Operator overriding 420.58 ms
Filter without count	0.01 ms	0.01 ms	0.01 ms
Filter with count	1,42 ms	1,98 ms	31,14 ms

The results of all test are presented in Table 2. From the results following conclusions could be made:

- Lambda expressions usage for sorting is fastest in C# and slowest in Python;
- Lambda expressions have the same or even lower performance for sorting then the case when comparer is used;
- Usage of comparator operators overriding is convenient but very ineffective operation in Python;

- Filtering operation as a separate operation is very effective in all compared languages;
- Additional operations on filtered data like transform it back to list are the slowest part of the filtering process. They are significantly slow in Python;

An extended research on the filtering options in Python language has been performed. The three additional techniques used for filtering are:

• **Filter with FOR loop:**

```
for x in emp_list:
if x.getSalary() > 50000:
emp_list_filtered.append(x)
```

• **Filter with List Comprehensions:**

```
emp_list_filtered = [x for x in emp_list if x.getSalary() >
50000]
```

• **Filter with External Function:**

```
def filter_salary(emp):
if (emp.salary > 50000):
return True
else:
return False
emp_list_filtered = list(filter(filter_salary, emp_list))
```

Also, filtering with inline lambda expression presented in the previous tests is considered together with these three techniques. Results are presented in Table 3.

Table 3. Filtering test results

Test	Python
Filter with FOR loop	27.51 ms
Filter with List Comprehensions	17.81 ms
Filter with External Function	19,43 ms
Filter with Filter function	31,14 ms

The results show that the optimal way for collection filtering is to use List Comprehensions. Almost similar behaviour could be accomplished using the **filter** function with external comparison methods. **FOR** loops and inline lambda functions present extremely low filtering performance.

4. Conclusion

In the paper is made an overview of lambda expression usage in some common programming languages and are presented results from tests about their performance.

All the test are performed over lists with 200000 objects. Sorting and filtering capabilities are tested using techniques with and without the usage of Lambda expressions.

It could be concluded that in most cases performance of lambda expressions are comparable to other programming constructions for the same tasks. However, lambda expressions have other advantages in addition to the reliable performance:

- Reduced syntax;
- Sequential and Parallel execution support;
- Higher Efficiency with parallel execution;
- Internal iteration of collections.

From the considered programming languages (C# Java and Python) C# has the best performance.

Another important result is from the speed comparison of different filtering techniques in Python. Tests show that for Python collection filtering is optimal to use List Comprehensions.

In future researches it will be important to extend the investigation of Lambda expressions capabilities in other programming languages. Also, it will be interesting to develop additional sets of collections and data structures that could be used as test data.

Acknowledgements

This research was funded by the National Science Fund of Bulgaria (scientific project "Digital Accessibility for People with Special Needs: Methodology, Conceptual Models and Innovative EcoSystems"), Grant Number KP-06-N42/4, 08.12.2020.

References:

- [1]. Slonneger, K., Kurtz, B. (1995). *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley Longman Publishing Co., Inc., USA.
- [2]. Fourment, M., Gillings, M.R. (2008). A comparison of common programming languages used in bioinformatics. *BMC Bioinformatics*, 9, 82.
- [3]. Kalemba, E., Ade-Ibijola, A. (2019). A Metric for Estimating the Difficulty of Programming Problems by Ranking the Constructs in their Solutions. *2019 International Multidisciplinary Information Technology and Engineering Conference (IMITEC)*, Vanderbijlpark, South Africa, 1-9.
- [4]. Cerveira, F., Fonseca, A., Barbosa, R., Madeira, H. (2018). Evaluating the Inherent Sensitivity of Programming Languages to Soft Errors. *14th European Dependable Computing Conference (EDCC)*, Iasi, Romania, 65-72.
- [5]. Donchev, I., Todorova, E. (2022). Dynamic Polymorphism without Inheritance: Implications for Education. *International Journal of Advanced Computer Science and Applications*, 13(10), 643 – 649.
- [6]. Urma, R., Fusco, M., Mycroft, A. (2018). *Modern Java in Action: Lambdas, streams, functional and reactive programming*. Manning.
- [7]. Nachenga, N. (2020). *Java Lambdas : Introduction to Java 8 Functional Programming*. Independently published.
- [8]. Mazinianian, D., Ketkar, A., Tsantalis, N., & Dig, D. (2017). Understanding the use of lambda expressions in Java. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), 1-31.
- [9]. *Java lambda expressions*. (n.d.) Oracle. Retrieved from: <https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html> [accessed: 17 May 2023].
- [10]. Langer, A. (n.d.). *Lambda Expressions and Streams in Java*. Angelikalanger. Retrieved from: <http://www.angelikalanger.com/Lambdas/Lambdas.html> [accessed: 20 May 2023].
- [11]. Uzayr, S. (2022). *Mastering C#: A Beginner's Guide*. CRC Press.
- [12]. Troelsen, A., & Japikse, P. (2021). *Pro C# 9 with .NET 5: Foundational Principles and Practices in Programming*. Apress.
- [13]. Sangle, S., & Muvva, S. (2019). On the use of lambda expressions in 760 open source Python projects. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1232-1234.
- [14]. Sharma, V.K., Kumar, V., Sharma, S., Pathak, S. (2021). *Python Programming: A Practical Approach*. Chapman and Hall/CRC.
- [15]. Rao, A. E., & Chimalakonda, S. (2020). An exploratory study towards understanding lambda expressions in Python. In *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, 318-323.
- [16]. Krishna, A. (2023). *Mastering Lambdas: A Guide to Anonymous Functions in Python*. Ashutoshkrris. Retrieved from: <https://blog.ashutoshkrris.in/mastering-lambdas-a-guide-to-anonymous-functions-in-python> [accessed: 12 June 2023].