Examining the Effects of Advanced Programming Constructs

Luca Davies
M.Sci. (Hons.) Computer Science (with Industrial Experience)

4th June 2021

Declaration

I certify that the material contained in this dissertation is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation. Regarding the electronically submitted work, I consent to this being stored electronically and copied for assessment purposes, including the School's use of plagiarism detection systems in order to check the integrity of assessed work.
I agree to my dissertation being placed in the public domain, with my name explicitly included as the author of the work.

Name: Luca Davies
Date: 01/06/2021

Abstract

As time has gone on, languages have become more developed themselves more advanced syntax constructs have been added alongside many "quality of life" type structures to make the writing code not only more efficient but also easier for developers. This report examines a number of these advanced constructs, drawing together both data from surveys taken by professional developers and data from analysis of a small number of large, active, open source code bases. Using a short but broad survey, data was collected on developers usage of these constructs and they provided information on how they felt these constructs affected their code and how often they used them, and how they thought they existed within the context of the wider industry. Analysis of code was carried using a combination of both simple regular expressions and via the user of lightweight tool to aid in the bulk processing of a large volume of source code file. It was found that ... don't know yet. FINISH ABSTRACT.

[Is there any working docs?]

Contents

1 Introduction

1.1 Overview

Programming as we know it today - in terms of high-level languages - has been around for over half a century. From FORTRAN in 1957, to Java in 1995, to Swift in 2014, the languages we use to power our ever-growing number of devices have been getting more and more powerful themselves. The tools we have now allow us to do things that would have been thought impossible 50 years ago. Programs that may have used 1000 lines to perform maintenance on a device in '60s, can now be written to maintain devices running thousands of times faster *and* take care of many many more tasks... all in the same number of lines. More and more we're also depending on our programming, a degree of code will be involved in nearly any electronic device you can use today, and not just those devices in domestic settings, but also those piloting our aircraft and cars and keeping our electricity network functioning.

It's easy to see why then, that the ease of *anything and everything* in programming is extremely valuable. There is thus an incentive to keep our programming languages simple. If a set a rules is very simple and easy to understand, it should be equally easy to abide by them and prevent mistakes - however, if that were the case, why would ever have left machine code or assembly language behind? They are simple, but that is also their downfall. Putting together a large simple program in this way become for too complex for the average programmer to keep in their head while programming, and there's no question that it's tougher to spot mistakes when your code is only half human-readable. So we can amend our axiom of simplicity to not be in the nature of the language itself, but in our interactions with the language: the easier we can make it to write our code, read our code, check our code, the lesser the chance of a mistake making it on to a production device and causing damage or harm.

This is exactly what we have seen happen over time. As languages do become more complex, the hope is that they become easier and easier for a programmer to use. Available syntax was once only of the form `command operand`, but now there's a myriad of complex but highly useful syntax and control flow structures to make the life of a developer much easier than it may have otherwise been. Though flow-control remains a key part of imperative languages, some declarative languages need not even define *how* something is done, simply *what* must be done. In imperative languages there are still abstractions of another type: those constructs that take existing syntax and functionality and compound it into a singular construct that programmers may use as a form of short-hand.

It is these constructs that this paper examines, using C#, JavaScript, and Java as a representative sample of popular imperative programming languages. Namely we will be focussing on the following constructs and their usage in these languages:

- Ternary/in-line If Statements (`a ? b : c`)
- Null-coalesce (`a ?? b`) and Null-conditional Operators (`a?.b`)
- Lambda Expressions and Anonymous Functions (`(a) => { b; }`)
- Additional constructs:
  - Foreach Loops (`foreach (a in b)` / `for ( a : b )`)
  - Unary Increment Operators (`a++`, `b--`)
  - Compound Assignment Operators (`a += 2`, `b -= 2`, `c *= 2`, `d /= 2`, etc...)

Detailed descriptions of these constructs are given in section 3

They may be used for multiple reasons, such as to make code simpler, or clearer. Other times, it may be style defined either by a programmer themselves, or the house rules of their organisation. No tool however it unable to be misused - overuse of these types of constructs or use of them in inappropriate places can in fact make code more complicated and harder to read than it would have been using more basic syntax. In this paper, we will study the above constructs and how they are used, with a view to making recommendations about they are best used, and where they are best avoided.

1.2 Motivation

The primary drive for this study came from my experience during my industrial placement (SCC.419). The codebase I was working on was extensive and developed by numerous developers over the course of

the last two decades or so. There were times that code was either made easier or harder not by its flow, but by the way it was written. The first instance that came to my attention was a ternary if-statement used that was so long that it needed to be split across four lines, had a second and third ternary in the true *and* false branch of the enclosing ternary, with lengthy expressions being evaluated from there. In this instance, it would have been significantly clearer to use a regular if-statement. Despite this the existence of ternary if-statements is still useful, but it brings to question where these constructs and other similar constructs should best be used.

There also seems to have been relatively little research into languages features such as that which will be examined here, either in terms of subjective readability or objective performance. It is hoped that by formally examining some of these constructs in a small selection of languages that the contribution provided will, if not self-contained, provide a launch pad for more thorough examination.

1.3 Aims & Objectives

The aims of this report are as follows:

- To identify and understand the usage of a selection of advanced programming constructs (as defined in sections 1.1 and 3.1)
- aha
-

1.4 Report Structure

The remainder of this report will discuss ...

2 Background

Some text

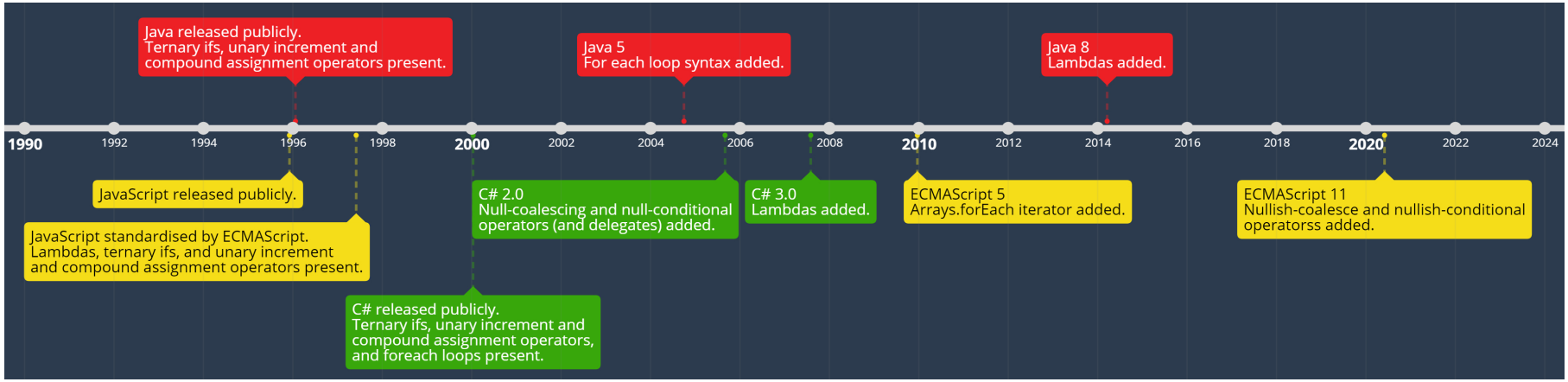

Figure 1: Timeline of the addition of advanced constructs to three modern languages.

3 Not sure

3.1 Constructs

Need this in detail somewhere.

- Ternary/in-line If Statements
  - Reduces the common `if ... else ...` pattern into a single line of the form `a ? b : c`, where `a` is a conditional expression, `b` is the body of the `if` and `c` is the body of the `else`.
- Lambdas and Anonymous Functions
  - A lambda expression is any expression involving a function used as an argument, just as in Lambda Calculus. Though this is not required, lambdas most often take the form of anonymous (unnamed) functions defined in-line. (Mazinanian et al. 2017)
- Null-coalescing Operator
  - Used to provide a default value in place of a null value. Replaces a null-check (using an if statement) with a single assignment of the form `a = possiblyNullValue ?? valueIfNull` where `a` is of a nullable type. (Gunnerson & Wienholt 2012)
- Null-conditional Operator
- Foreach Loops
- Unary Increment Operator
- Compound Assignment Operators

4 Survey

[]

5 Code Analysis

[]

5.1 Sample Code

[]

- Picked up basic for in place of JS for-to (not for/each)

5.2 Real-World Code Bases

[]

6 Conclusion

[]

6.1 Review of Aims

The following list repeats the aims presented at the start of this report, with each aim followed by an overview of relevant results obtained.

- // []
- // []
- // []
- // []
- []

6.2 Reflections

[]

6.3 Negative Impacting Circumstances

The on-going coronavirus pandemic is still very much a factor in the lives us all. :(

6.4 Future Research

[]

6.5 Closing Statement

[]

References

Gunnerson, E. & Wienholt, N. (2012), *A Programmer's Guide to C# 5.0*, Apress.

Mazinanian, D., Ketkar, A., Tsantalis, N. & Dig, D. (2017), 'Understanding the use of lambda expressions in java', *Proc. ACM Program. Lang.* **1**(OOPSLA).
URL: *https://doi.org/10.1145/3133909*

Appendix

Appendix 1: []

[]

Appendix 2: []

[]