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` / `a?[x]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 **??**

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)
- To collect data from a number of professional developers about when and how they think it is appropriate to use these constructs
- To examine what guidance is readily available to developers in typical CI/CD toolchains
- To examine a small number of large open-source repositories to assess whether these constructs are used consistently in practice

1.4 Report Structure

The remainder of this report will discuss ...

2 Background

Some text

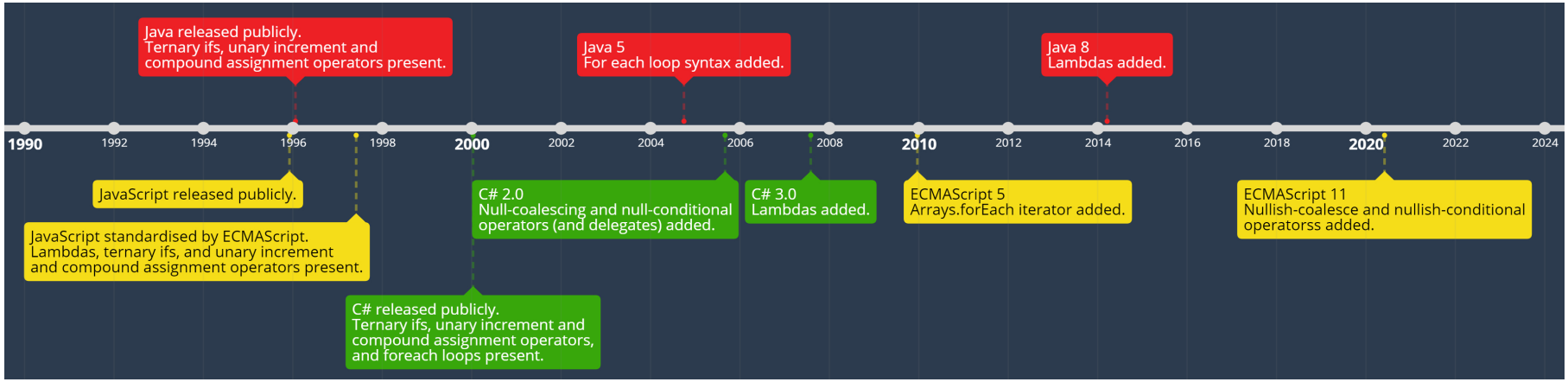

Figure 1: Timeline of the addition of advanced constructs to three modern languages.

3 Study Design

3.1 Constructs

The section will give a more detail description of each of the constructs to be examined. The constructs were selected as they are all replaceable with simpler syntax (in most scenarios - lambdas being a occasional exception, see section **??**) and, together, they represent a range of commonality from long-used compound assignment and unary increment operators at one end of the scale to much newer additions such as the null-coalescing and -conditional operators. The constructs are as follows:

- 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`. While it may be possible in some languages to have multiple statement in the 'b' and 'c' sections of a ternary, but standard syntax only permits a single statement.
- 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
  - Used to access a member or element of its operand *if and only if* that operand evaluates to non-null. Otherwise, it returns null. (Various 2015). This allows a member/element access operation to be carried out without a null-check on the base operand, only on the evaluated value.
- Foreach Loops
  - Used to simplify for loops for the purposes of iterating over an abstract data structure, usually a type of some form of collection of like objects; those that can be indexed into to retrieve an element. For example, in C#, a `foreach` loop may be used to iterate over the elements of an instance of any class that implements the `IEnumerable` interface. The same applies to Java with its `Iterable` interface. In C#, if `numList` is of type `List<int>`:

    ```
    for (int i = 0; i < numList.Count; i++)
    {
        total += numList[i];
    }
    ```

    Can be written as:

    ```
    foreach (int num in numList)
    {
        total += num;
    }
    ```
- Unary Increment Operator
  - Used to increment or decrement an integer variable by one. Often seen in for loops to increment the index variable.
  - For example:

    ```
    a = a + 1;
    ```

    Can be written as:

    ```
    a++;
    ```

- Compound Assignment Operators
  - Used to effectively "append" to a variable's value, compressing three references to variables/literals into two. In most supporting languages, a compound assignment operator exists for all basic arithmetic operators, logic operators and some others (such as in string concatenation).
  - For example:
    ```
    a = a * 2;
    ```

    Can be written as:
    ```
    a *= 2
    ```

3.2 Survey

One of the facets of this study was to capture and analyse the perceptions and thoughts of a number of professional software developers with regard to the constructs being examined. This was carried out using a survey, constructed following guidelines laid down by both published literature (Rowley (2014), Driscoll (2011)) as well as a small amount of grey literature provided by higher education bodies.

3.2.1 Questions

Prior to designing, it was established that the survey must be able to provide insight that may help to answer the following questions:

1. Are the participants *aware* of the constructs?
2. Do the participants *use* the constructs?
3. Are participants *encouraged to use* the constructs?
4. Do the participants think these constructs are better for:
   (a) brevity?
   (b) clarity?
   (c) code efficiency/performance?
5. Do the participants think that the constructs are better used in some languages than in others?
6. Do participants rewrite code *to use* or *to not use* the constructs?

Hypothesis/expected results

3.2.2 Participants

Participants were recruited passively from within the Information Systems Services department (ISS) within Lancaster University by means of a blanket email sent out to all teams and filtered down via regular communications channels within the department. This gathered 40 responses in total, 23 of which completed the survey and may be considered valid responses.

Due to the distribution method, there is no guarantee that all respondents are software developers by profession, however, all participants had a minimum of one year of programming experience, nearly half falling into the 1-4 years of experience category, with three participants having between 10 and 14 years experience and the remaining 8 having over 20 years of experience. This statistic was the only personal data collected and does not identify participants.

3.2.3 Questions

The survey was created using Qualtrics, as provided by Lancaster University, and consisted of a maximum of 16 questions and a minimum of 2 questions (depending on certain answers). Participants were first asked to indicate which of the constructs they were aware of so that only questions and option relating to those constructs they were aware of were shown to them. For each construct they were of, participants were asked to a series of questions design to address the questions listed in section 3.2.1, focussing on: frequency of usage, reasons to and to not use each construct, perception on code performance, and perception of industry-wide usage per-language.

Lastly, participants were asked if they had any further comments, which invited some very interesting points that will be discussed alongside the rest of the results in section 4.1.

3.3 Static Code Analysis

Static code analysis is concerned with checking programs and code for errors without the need to actually execute it. That is, a static code analysis tool will read the source code (or the compiled intermediary language in some cases), construct some form of abstract model of the program and then run a series of tests (often pattern-matching in nature) to detect well-known possible pit-falls, bugs, and problems Louridas (2006).

These tools will often make recommendations on how source code may be improved in numerous way beyond explicit bugs and problems, such as stylistic improvement that make use of advanced syntax whatever given language is being analysed - advanced syntax akin to the constructs being examined here. Accordingly, static analysis tools were employed to investigate whether or not they made any recommendations for or against any of the constructs in question. SonarQube and Semgrep were selected for this purpose as they both support all three languages being subject to analysis, they are both under recent, active development, and because they both have free variants. Both tools were run against sample control code files and large open source repositories in the earlier mentioned three languages: C#, JavaScript, and Java.

3.3.1 Sample Code Files

The sample code files were created to act as a baseline or control test to see if SonarQube or Semgrep made any note at all about the bare usage and/or inclusion of any of the constructs.

Each file contains a bare (non-contextualised) example of each of the constructs placed directly alongside their 'simple syntax' functionally equivalent counterparts. Each file is valid syntax for its language, and all of them may be executed, though none of them have a true entry point or way to actually *run* the code within itself - the functions and methods are present purely to *be present* so that they may be analysed by SonarQube and Semgrep. Though there are differences between the three languages, the form of the sample code was kept as similar as possible to preserve their purpose as control samples, without adding deliberately unusual syntax that may contaminate the output of SonarQube and Semgrep.

3.3.2 Open Source Repositories

Four large, open source code repositories were selected from GitHub to be analysed for this study, one each in C# and JavaScript, and two in Java. The selection process was simple:

3.4 Git Commit Analysis

To supplement the static analysis of the repositories, manual, by-hand analysis of the commits made to these repositories was also undertaken.

The goal of this branch of the study was to understand if and how standards are being maintained within the repositories within reference to the constructs and to general style-keeping. This was carried out by cloning the repository and examining the commit history using `git log`. A list of keywords were run against the commit history using the `--grep=<pattern>` option to `git log`. The keywords used are listed below:

General	style
Ternary	ternary, conditional, `?:`, elvis
Null coalesce/conditional	null, coalesce, conditional, `??`, `?.`, `?[`
Lambda	lambda, arrow, `=>`, `->`
For each	foreach, for each
Unary Operators	unary, increment, decrement, ++, --
Compound Operators	compound, assign, `+=`, `-=`, `*=`, `/=`, `|=`, `&=`

Some of these keywords generated cross-over between those meant to highlight changes around a particular construct, but this was unimportant as these keywords were only used to create a list of 'interesting' commits. Commits were deemed interesting if the commit message and/or description contained reference to any of the keywords in such a way that it seem plausible that a change was made regarding any relevant construct. For each commit in the resultant list, the changes were scrutinised closely to pick out changes made that: added, removed, altered, or commented on usage of any of the constructs. Each type of change was documented and counted per commit.

4 Results

4.1 Survey

[]

4.2 Static Code Analysis

[]

4.2.1 Sample Code Files

Bugger all picked up, tools do not care

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

4.2.2 Real-World Code Bases

[]

5 Conclusion

[]

5.1 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.

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

5.2 Reflections

[]

5.3 Negative Impacting Circumstances

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

5.4 Future Research

[]

5.5 Closing Statement

[]

References

Driscoll, D. L. (2011), 'Introduction to primary research: Observations, surveys, and interviews', *Writing spaces: Readings on writing* **2**, 153–174.

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

Louridas, P. (2006), 'Static code analysis', *Ieee Software* **23**(4), 58–61.

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*

Rowley, J. (2014), 'Designing and using research questionnaires', *Management research review* .

Various (2015), *C# 6.0 Draft Specification*, Microsoft Corporation.

Appendix

Appendix 1: []

[]

Appendix 2: []

[]