Examining the Usage 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 seven of these advanced constructs, drawing together both data from surveys taken by professional developers and data from analysis of a four widely known, 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 WITH RESULTS.

[Is there any working docs?]

Contents

1 Introduction

1.1 Overview

Code style has long been a topic of heated debated within the field of computer science since even the very early days. You will rarely ever get a single consensus in room when asking "How should code blocks be indented?", or worse still "What is the best brace style?". These kinds of physical layout questions have been around as long as the languages they exist within, but as programming languages have become more complex and afford programmers greater ease to express their logic, this discussion of best style has only broadened. Languages include more 'syntactic sugar' constructs, patterns and operators that allow the simplification of previously more complex-looking or verbose statements. Such constructs and operators as:

- 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 2.2

Prior to this project, I was working on a large C# codebase with a nearly 20 year history. Throughout that time, the notion of 'ideal' style had changed not only within the company the codebase belonged to, but in the wider software industry too. This led to wide mix of 'plain' (or without syntactic sugar-like statements) together with many of the newer syntax constructs. For example, lambdas were only added to C# around seven or eight *years* after the first iteration of the codebase had been written. Some files featured a complete lack of any of these more advanced syntax constructs, some had many, some had some very painful to read examples of their use, where perhaps it may have been better to have used more traditional syntax. It is this mix-and-match of the use of these constructs that spurred the idea of this project. Looking at code, there are clearly times when traditional syntax is overly verbose and takes up more space on the screen than necessary, but it also became abundantly clear that to swap out *all* plain syntax for these more compact styles would be a big mistake, making many bits of code much harder to read.

So where does the line lie between when to, and when to not use these constructs? To answer this, we must consider why we would use either one or the other at all. Often they are used to make code more compact - as software development has grown as an industry, there are certain 'set' statements that it can generally be assumed that everyone understands, and that can be abstracted a little to make them neater or smaller. Other times it may be what has been laid out in a company style guide that everyone should use, or perhaps purely the habits and practices of an individual. However, regardless of the exact reason or motivation behind every use of these constructs, it is important to remember that no tool is unable to be misused - overuse of these types of constructs or use of them in inappropriate places, can make code more complicated and harder to read than it would have been otherwise. It is equally important to value the 'plain' or verbose syntax as much as the compact variants, both having very unique and distinct advantages and disadvantages.

This project began with an aim to make recommendations on how to best use the constructs in question, but as research continued and it was found just how little clear guidance is available, and how much conflict and indecision is present, it has become a more exploratory study in the use and usage of theses constructs. To do so, we focus on C#, JavaScript, and Java as a representative sample of popular imperative programming languages.

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. 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 stems from my experience during my industrial placement (SCC.419). As briefly mentioned above, 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 can be assumed that 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 these, either in terms of subjective readability or objective performance. Some studies I have come across in research even make explicit note of the fact their definition of 'good style' is subjective and opinion-based, albeit from that of perceived experts. Using experts in a field is always a good sign, but if there really is very little guidance to any level of programmer, then it stands to reason that much of what even experts think is good style, could be from personal experience alone, or at least somewhat unacademic.

It is hoped that by formally examining these constructs in an academic context that the produced research will a launch pad or grounding for more thorough examination across the software development industry as a whole.

1.3 Aims & Objectives

The aims of this report are as follows:

1. To understand the usage of seven advanced programming constructs (as defined in sections 1.1 and 2.2)

2. To study what reasons developers have to, or to not, use these constructs

3. To make take the first steps in creating formal, unified recommendations on the use of these constructs

These aims will be achieved via the following objectives:

1. Collect data from professional developers about when and how they think it is appropriate to use these constructs via a survey

2. Investigate what guidance is available to developers on how to use these constructs by examining available style documents provided by developers and online

3. Examine the recommendations made by static code analysis tools on specifically written 'control' code files

4. Analyse four open source repositories to assess how these constructs are used in practice both by hand and using static code analysis tools

5. Conduct profiling and performance tests of the constructs (stretch objective)

1.4 Methods

A multifaceted approached was taken to achieve the above aims, incorporating multiple methods. First, a survey was carried out, to gather both qualitative and quantitative data direct from developers about how they use each of the constructs being examined - this helps address aim one. A survey was chosen as it provides a comparatively easy method to gather answers to base questions from a large sample size. Quantitative data was collected, with regard to aim two, by employing static code analysis tools to analyse four open source GitHub repositories and 'control' files (giving examples of both the constructs and their 'plain' equivalents), in three programming languages. Static analysis tools were employed as they most likely represent actual tools developers may use to check their code, and thus the rules and recommendations that will be levelled against their code in a real scenario. Aims one and two were also supported by the analysis of the commit history of the aforementioned four repositories for changes that include or exclude the use of the constructs in question. Lastly, aim three will be met via a culmination of all methods used, and some simple numerical analysis.

1.4.1 Threats to Validity

Due to time constraints and limits of contacts, the survey was only able to encompass one group of programmers, all from within one company. Although the technology stack the developers in question may have been exposed to consisted primarily of C# and JavaScript, this potentially means participants may have been answering questions with little knowledge on Java. The results of the survey are also difficult to generalise, as there are many factors which may affect responses that are limited to this one group of developers.

The survey was distributed widely within ISS, giving approximately 60 potential respondents. Of those, there are 44recorded responses (70% response rate), 23 of which are completed, valid responses (40% valid responses).

1.5 Report Structure

The remainder of this report will first detail further background research directly and indirectly related to this topic, before describing the methods and design of the study, followed by the results and information gained from the study, discussion on these results, before final conclusions.

2 Background

At face-value, much of the grounding of this study may be considered conventional wisdom. That is, the constructs are simply used anywhere and everywhere according to when we as programmers deem it appropriate, without much prior thought or formal procedure. This once would have been the case for structural or physical style to our code, the way we lay our code out and arrange it around whitespace. Countless style guides have concerned themselves with this for many years, and there's still much debate around which brace style is the best in C-like languages, or how many spaces should be used to indent code blocks. It hasn't been until much more recent times, as software development houses have grown, and the computing industry as a whole has ballooned into the incredibly important sector it is today, that we have started to think more about the meta-style of our programming. Not how it's physically laid out, but how we layout our logic and design patterns. Somewhere between these two schools of thought lies the area this study is focussed on.

It does not appear, however, that a great deal has been written and published around how to best use many of the programming constructs that many programmers will use tens, if not hundreds, of times per day. What follows is a summary of much of the existing works that cover partially on wholly the same topics.

2.1 Existing Literature

As far back as the 1980s, Berry & Meekings (1985) proposed the Berry-Meeking Style Metric as a measure of good style and readability of code. This used style analysers adapted specifically to C to produce a metric denoting the 'lucidity' of the code. The metric was derived from 11 different characteristics about a program: module length, identifier length, percentage of comment lines, amount of indentation, percentage of blank lines, line length, spaces within lines, percentage of constants, reserved words used, included files and number of gotos used. Zooming out from this level of metric-based scoring, Oman & Cook (1988) collects a number of studies of together and notes how application of these types of style metrics not only lack correlation with error proneness, but also has unpredictable effects on complexity metrics. They go further to say that style is an 'intuitive and elusive' concept, that is 'highly individualistic' and simultaneously easy to see, but hard to define. Many differing style guides are either too general or subjective, contradictory between each other and with no guidance and how to manage these conflicts. They go on to liken the use programming style to "the 'perception and judgement' a writer exercises in selecting 'from equally correct expressions, the one the best suited to his material, audience and intention'". This is to say that the nuance of writing in English prose is present too in the style of code.

In more recent times, it is again reinforced that some simple metrics like program length cannot solely be used to measure the quality of code style. Moghadam et al. (2015), when showing a range of students' responses to a programming task (the shortest of which is unlikely to be rated the easiest to read), state that length is not a sole indicator of good style, and moreover that "excessive terseness is often worse than that of verbosity". We still don't have any *definite* way to judge 'good' style, with Roy Choudhury et al. (2016) noting that their study on providing style hints to new programmers is limited by the fact the given 'good' style is subjective and opinionated.

One of the defining style guides for Java, Effective Java (Bloch 2018) makes many assertions about how to use certain features of the Java language, but most of these do not reach the level of granularity focussed on in this study. The reference to anything close is that Java programmers should "prefer for-each loops over traditional for loops". Lambdas feature in their own chapter, but only with reference to *how* to use them, not *when*.

It is reasonable to think that official style guides published by programming language developers would be cover-all for all sorts of style, from the extraneous and finnicky to most basic syntax, but in fact, Microsoft's own style guide for C# is somewhat short, and it's only relevant mention for this study being a note that lambdas are 'shorter' and may thus be good substitutions for the more verbose regular methods that would be needed otherwise.

Two prevailing style guides for JavaScript have come into prominence the last few years, both with certain small conflicts and differences from each other. These are the Google style JavaScript style guide (Various n.d.), and the Airbnb JavaScript style guide (Various 2021). The Google guide advocate for use of `for-of` loops over all other types of `for` loop in JavaScript, allowing `for-in` loops exceptionally for dict-style objects. The Airbnb guide, conversely to Google, requires use of *iterators* over `for-in` or `for-of`, with no reference to standard for loops. It also makes one very relevant assertion: that ternaries

should "not be nested and generally be single line expressions". Perhaps controversially, the Airbnb guide also rejects the use of the unary operators, `++` and `--`. This is primarily due to the fact that these statement cause automatic semicolon insertion and can cause silent errors. However, it is also reasoned that the more expressive `+=` and `-=` operators allow code to be easier to read and less error-prone.

2.2 Constructs

This section will give a more detailed description of each of the constructs to be examined. The constructs were selected for a number of reasons - firstly, they are all replaceable with simpler syntax (lambdas being a occasional exception, see Section 5.2.1), conventional wisdom would dictate that some are rather commonly used (compound assignment / unary operators) and conversely that some are used much less (null-coalesce / null-conditional). Initial thoughts of some of the constructs came about from seeing then during my work for SCC.419. The constructs in question are detailed below:

- 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).
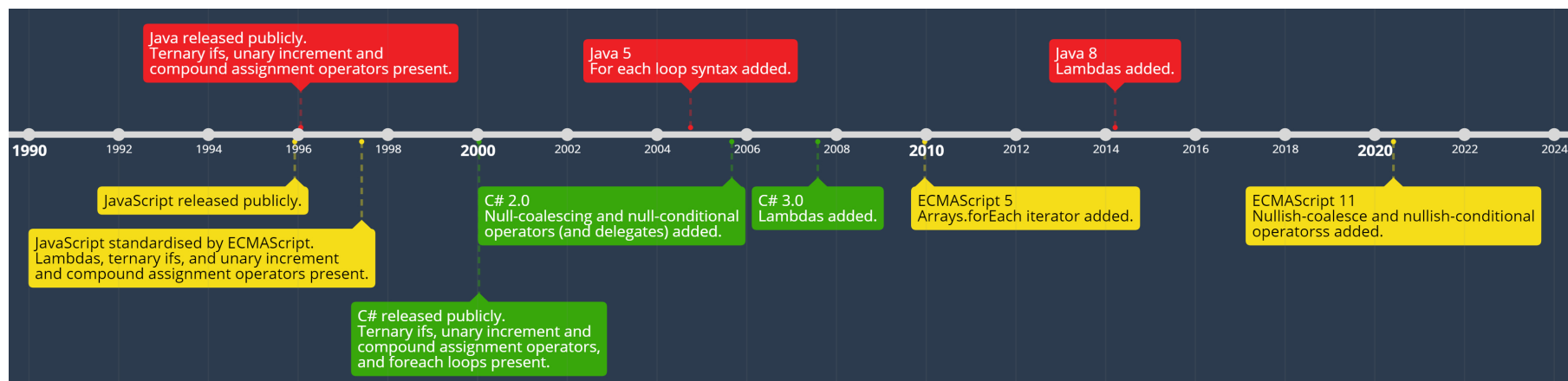
  - For example:
    ```
    a = a * 2;
    ```

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

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

3 Study Design

3.1 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. This allowed better questions to be written, that ensured they were no longer than necessary, used no double negatives, were not leading questions, only attempted to address one issue, and were not too vague. The design was also guided by these sources to include at least one open-ended question to allow participants to put forward any exceptional or unexpected information that could prove highly valuable to the study. Rightfully, notable mention is made at all stages to use of appropriate moral and ethical frameworks. In this instance, very little in the way of any problematic material is examined. The most personal question asked in the survey is how many years participants have been programming for - no other questions of a remotely personal nature are asked.

Using the Qualtrics platform also allowed the usability of the survey to be examined by the Qualtrics' inbuilt system. This advised the design of the survey to be as easy to use as possible, increasing the likelihood of respondents completing their submissions. The only exception to this was the use of matrix table questions, which are not particularly friendly to mobile users. Despite this, they were used as they provided the best and most concise way to display the literal matrix of options available due to the presence of multiple constructs the question pertains to. Figure 2 shows one such question. The full survey is available in Appendix 5.5

Figure 2: A matrix table question used in the survey.

3.1.1 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?

3.1.2 Participants

A convenience sample of participating developers was recruited 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.

3.1.3 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 3.1.1, focussing on: frequency of use, 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.

3.2 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.

3.2.1 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 use 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.

3.2.2 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:

3.3 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 use of any of the constructs. Each type of change was documented and counted per commit.

4 Results

4.1 Survey

As described in Section 3.1, a survey was conducted with experienced professional software developers and programmers. This revealed interesting patterns of usage (or lack of usage) of the constructs, and some interesting perceptions that may prove useful in future research focussed on performance and use in the wider industry.

The following sections will detail the results of the survey, including threats to the usefulness of the survey and the patterns discovered by the survey. All percentage values given have been rounded to 1 decimal place.

4.1.1 Threats to Survey

As mentioned in the introduction, the survey participants were only a convenience sample of developers. This means that the results may not be generalised very easily to a wider group of developers. The results may be generalised *within* ISS but not outside. Further research and more significant surveying should be undertaken to verify if the results found within ISS may be applied to a wider populace of developers.

Despite this, a survey still provides a very quick and easy way to gather information from a sample size much larger than would be trivial than many other methods.

4.1.2 Awareness

Awareness of the constructs was gathered by a simple multiple choice question with a binary yes/no for each. Participants could indicate whether or not were familiar with all, some, or none of the constructs.

All participants were aware of unary operators and ternary/in-line if-statements, nearly all were aware of compound operators and lambdas, and most were aware of null-coalesce and null-conditional. Figure 3 shows these results in a graphic format.
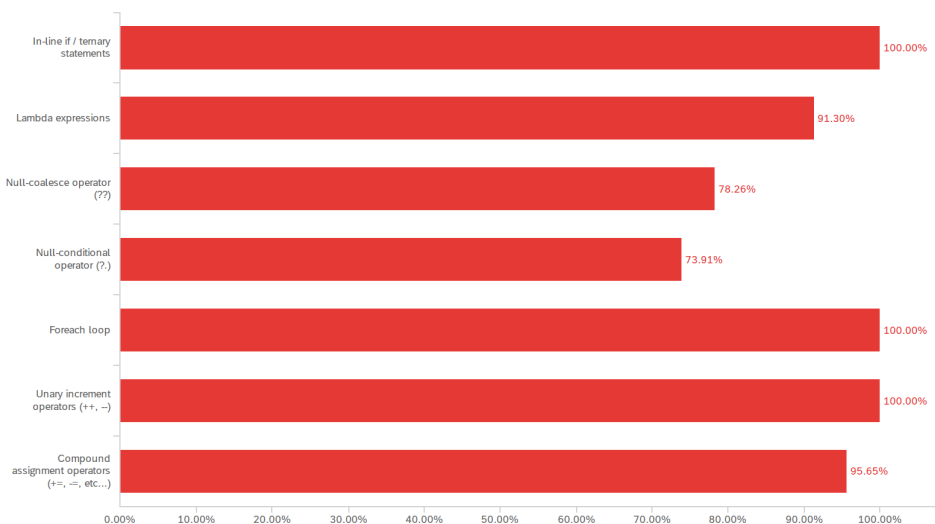

Figure 3: Percentage of participants aware of each construct

The most stand-out piece of information here is that the *absolute* awareness of the null-coalesce and -conditional operators was markedly higher than might be expected. This can likely be attributed to the fact that participants were all from within ISS, where the technology stack contains a significant amount of C#, in which these operators are often seen. If the survey were repeated over more diverse and broad range of developers working in more varied technology stacks (for example, those focussed much more on Java, or C/C++), it is reasonable to assume that awareness of these operators would be less. This is a trend that is repeated elsewhere in the results set. It is also of note (if not significantly so) that not *all* participants were aware of compound assignment operators, whereas all participants *were* aware of ternaries - a ranking inverse to what was expected. However, the difference in the numbers are not

statistically significant, comprising no more than one or two responses. To draw any conclusion from this datum would require it being replicated in a greater sample set.

4.1.3 Frequency of Use

To assess how often participants use each of the constructs, they were asked specifically: *In places where they can be used, how often do you use each of these constructs in your code?*, that is, "for every chance you *could* use the construct, how often do you?"

It was the extreme ends of the scales in this question that provided the most interesting results here. Firstly, and perhaps reassuringly, only one data point was recorded, for any construct, under the 'never' option, with just one respondent indicating that they never use the null-coalesce operator. Conversely, 'for each' was the most selected construct in the 'always' category, with 44% of participants selecting this option, followed closely by compound assignment operators at 41%.

Across all constructs, the majority of responses fell into the 'frequently' category, holding 39% of the total selections, with the 'sometimes' category at 27%. This general erring toward the more frequent end of the scale holds not only for the collective dataset but also for each construct when examined individually, as may be seen in figure 4
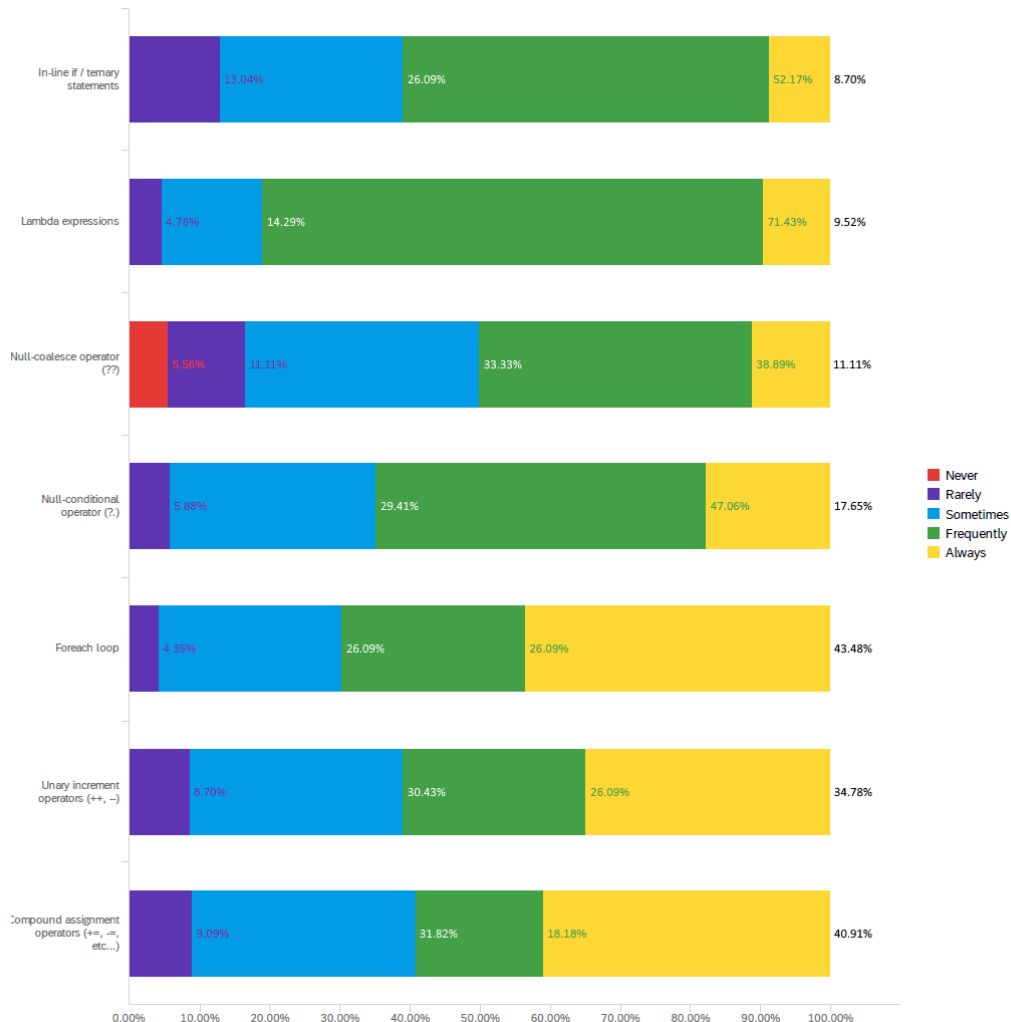

Figure 4: Frequency with which participants said they used advanced constructs, grouped by construct. (n.b. figures normalised to percentages)

As is visible in figure 4, the use of lambdas in moderation is the most agreed upon data point here, with just over 70% of responses labelled 'sometimes' for lambdas. This is likely explainable by a variety of reasons, not least is the fact that lambdas are very powerful and complex structures, much more

16

so than the other constructs being studied, and as such, the number of situations where they *can* be used likely greatly exceeds the number situations where they *should* be used. This distinction around lambdas becomes more apparent in later questions. Of note, again, the results for the null-coalesce and null-conditional operators. Although the numbers of participants familiar with these operators is fewer, those that were aware of them do not shy away from them, using them with comparatively similar frequency to the other constructs.

4.1.4 Reasons To Use

Participants were then asked to indicate *why* they chose to use each given construct. The four options were presented:

- "It is company style"

- "It makes code clearer"

- "It makes codes simpler"

- "It makes code run faster"

One or more of the above options may have been selected, with an exclusive option: "None of these" present too. If this was selected, participants were prompted to give their own reason.

The trend here was extremely clear: the most common reasons for using any of the constructs was for clarity and simplicity. "It makes code clearer" and "It makes code simpler" took 39.4% and 42.3% of all selections made respectively. The data for the selections totalled across each construct is shown in Figure 6.

Six participants selected 'None of these' for at least one construct, two of whom selected this option for all constructs they were aware of and both instead said that they choose to use the constructs based on personal preference or habit. Another two selected this option for lambdas alone, one saying that they are aware of don't use lambdas, and the other highlighting a possible flaw in this study regarding lambdas. The participant noted that there are places where "there is no rational alternative to using a lambda expression. It does not make the code simpler or clearer, it's just **necessary** to use a lambda." This brought to attention the fact that lambdas may be a more functional language feature, not fitting with the other constructs here. This is discussed further in Section 5.2.1. One participant also said that they would generally use 'map' instead of a `foreach` loop, claiming "it helps avoid 'off by one' errors". Off-by-one-errors occur when iterating over an array or other list of m through n items. Asking how many items there are (how many loop iterations are required), the intuitive answer of m - n is incorrect, the correct answer being $(m - n) + 1$. This is illustrated using a specific type of off-by-one error, a *fencepost error*, in Figure 5.
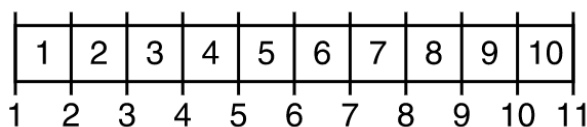

Figure 5: If you have a fence with 10 posts, how many bounded sections exist between the posts? The intuitive answer is 10, but it is in fact 11. This is a specific off-by-one error.

It is not clear what advantages a 'map' function has over a `foreach` loop to avoid off-by-one errors. In fact, conventional wisdom dictates that `foreach` loops, by nature should, avoid these errors by virtue of the fact that they *remove* the need for a programmer to calculate the necessary number of loop iterations. It is plausible to suggest that this response was given by mistake, instead meaning to refer to traditional `for` loops, rather than `foreach` loops.

Figure 7 part repeats the data above, again showing that clarity and simplicity are the most common reasons. We can also see that the split is *mostly* even between these two reasons, null-coalesce being a slight exception, with nearly double as many responses selecting simplicity over clarity. Null-conditional follows this pattern, but the difference is less pronounced. Other notable statistics are: no participants said that they used null-coalesce or -conditional due to company style, and the three constructs that had code run speed selected. Two participants selected this for lambdas, and one each both selected this for `foreach` loops and unary operators. These are statistically insignificant, but it is still of note that none of the other constructs had any selections regarding faster code running.
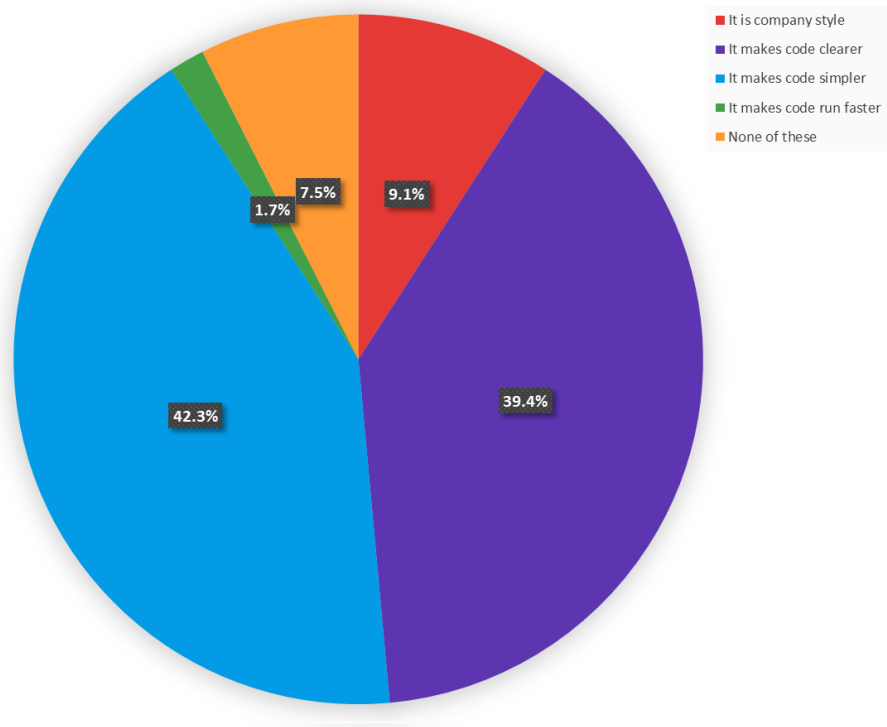
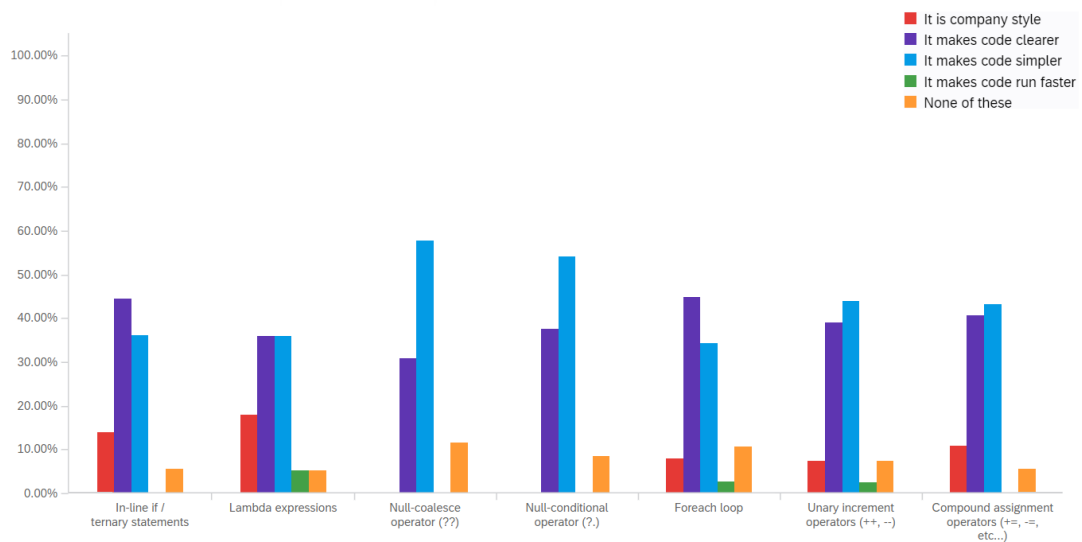*Figure 6: Reasons to use constructs, totalled across all constructs.*

Figure 7: Reasons to use constructs grouped by construct. (n.b. Figures normalised to percentages per construct)

4.1.5 Reasons To Not Use

To provide the other half of the story to the previous section, participants were also asked about the reasons they choose to *not* use the constructs. Similar to before, the options presented were:

- "It is company style"

- "It would make code less clear"

- "It would make code more complex"

- "It would make code run slower"

As before, one or more of the above options may have been selected, with a "None of these" option that if prompted participants to give their own reason if selected.

Reasons of clarity and complexity were the most selected again, with 48.2% of all selections attributing maintaining the clarity of code as a reason to not use a given construct, and complexity at 23.9%. More than double the proportion of selections were present for "None of these" compared to the when asked for reason *to* the constructs - here, 16.2% of selections were placed here. The reasons provided by these respondents were quite varied but very interesting and are all worthy of discussion:

- Personal preference

    - Mentioned explicitly by three, and implicitly by one, of the ten individuals who gave their own reasons.
    - One noted that "as long as code is well documented you can use any of them".
    - One wrote that there is no company style for these constructs and that there is instead guidance on larger/less granular things like examples of a model class. This is contradicted by other responses, though it's plausible the two are subject to different standards.
    - The third, the only of the ten that overlap with those that selected this option on the previous question. Simply stating that it is personal preference.
    - The final reason links usage of compound of unary operators goes against their preference to use a 'functional style' and to 'avoid mutability of variables'. This raises interesting questions about the use of functional programming ideas in non-functional paradigms.

- 'I [almost] always use this construct'

    - Two participants said they would always use unary increment operators
    - One said they would always use compound operators
    - Another said the only time there would not use null-coalesce/null-conditional is when null checking is not required

- Other/Unqualified

    - One participant stated that a `for` loop may be preferable to a `foreach` loop when the index variable is needed. They then go on to say that it also 'depends how I'm feeling'.
    - Regarding lambdas and unary and compound operators, one participant said: "sometimes these operators don't account for requirements such as type checking or formatting" without any more description.
    - One participant claimed that `foreach` loops may introduce errors, but did not explain further.

Only 4.5% selected "It would make code run slower" - logically, this would make sense factually too, as most of these constructs are plain syntactic differences that, with most modern compilers and interpreters, are likely to produce the same bytecode and/or machine code. Lambdas and `foreach` are likely the only exceptions to this. The former has other potential implications beyond just how it looks on-screen, the latter (depending on language) may be a change from an array index to initialising an iterator and calling on that instead. This is reflected in the thoughts of participants in Figure 9, showing that these two constructs were the only two to have received the selection pertaining to code run speed.

The breakdown in Figure 9 also confirms that clarity was the most common reason selected by a wide margin for four of the constructs, but not for unary and compound operators (multiple reasons,
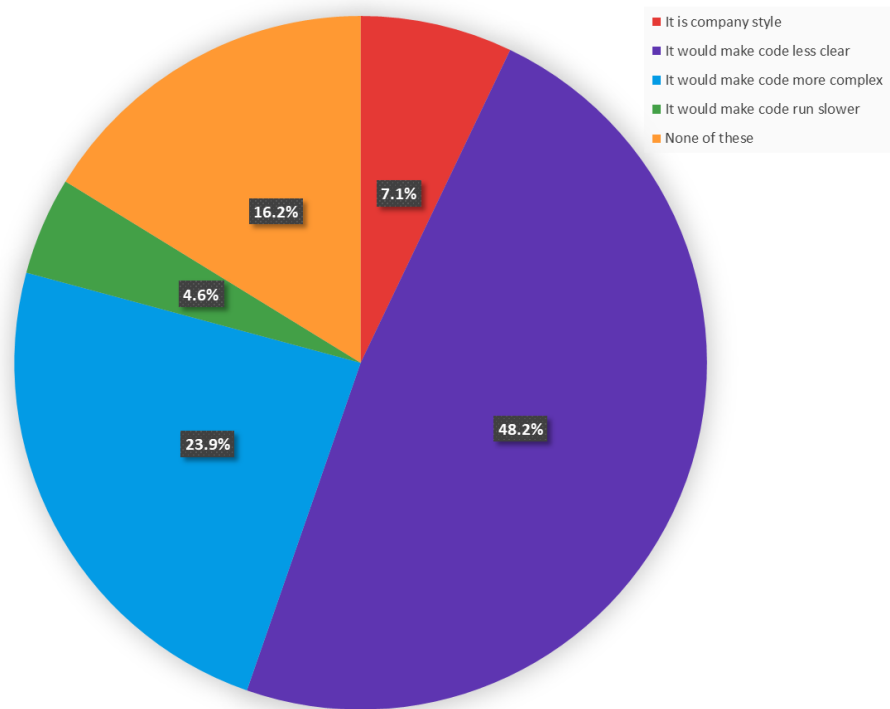
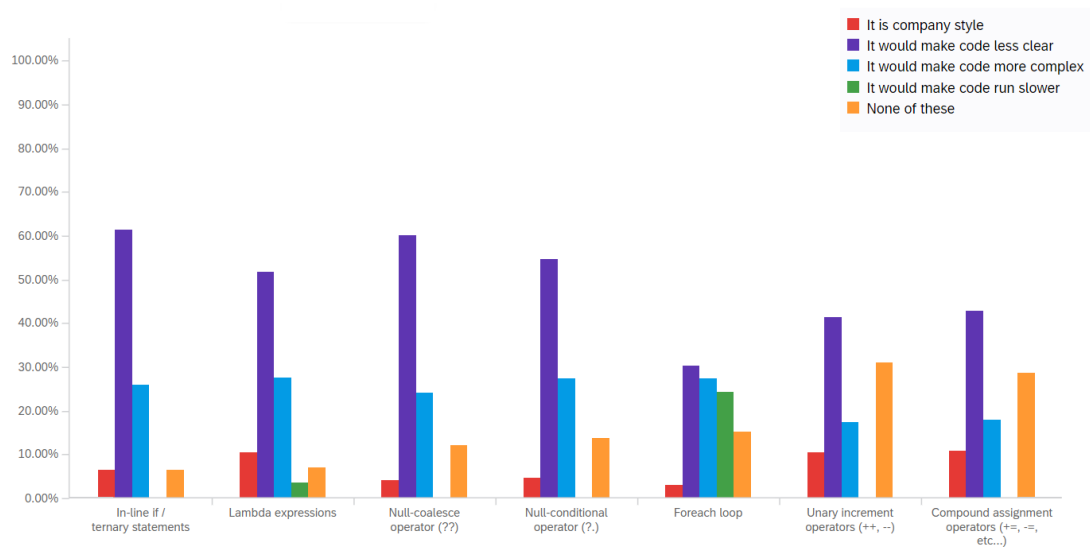Figure 8: Reasons to not use constructs, totalled across all constructs.

Figure 9: Reasons to not use constructs grouped by construct. (n.b. Figures normalised to percentages per construct)

described above) and for `foreach` loops (run speed was a concern as described immediately above). All constructs had a small number of responses claiming company style as reasons for not using them, but this is a relatively small number compared to the proportion that cited clarity or complexity or their reason.

4.1.6 Performance

As touched on previously, it was not expected with most of these constructs for there to be much of a bias with regard to performance (this is examined crudely in Section 4.2.3). Participants were asked to state what, if any, effect they thought each construct would have on the performance of code.

In line with expectation, a large majority, 61.2%, of the selections here were that of no change to performance. The second largest group was that of "Don't know" with 22.4% of selections. Both these statistics are shown in Figure 10. It is likely that most high-level programmers (such as those using high-level languages like C#, JavaScript, and Java0) either do not routinely give much consideration to such fine-grained efficiency, or are not concerned with relatively small performance variations that come from using these constructs.

The individual performance data also matches participants selections from previous question when grouped by construct: the majority for each is still "No change", but the stand out constructs that have a significant number of selections that are of increased or decreased performance are lambda expressions and `foreach` loops. Lambdas had the smallest proportion of selections for "No change", and the largest for "Increased performance", meanwhile `foreach` had the second smallest proportion of "No change" and the largest for "Decreased performance". The possible decrease in performance using `foreach` was discussed in the previous section.
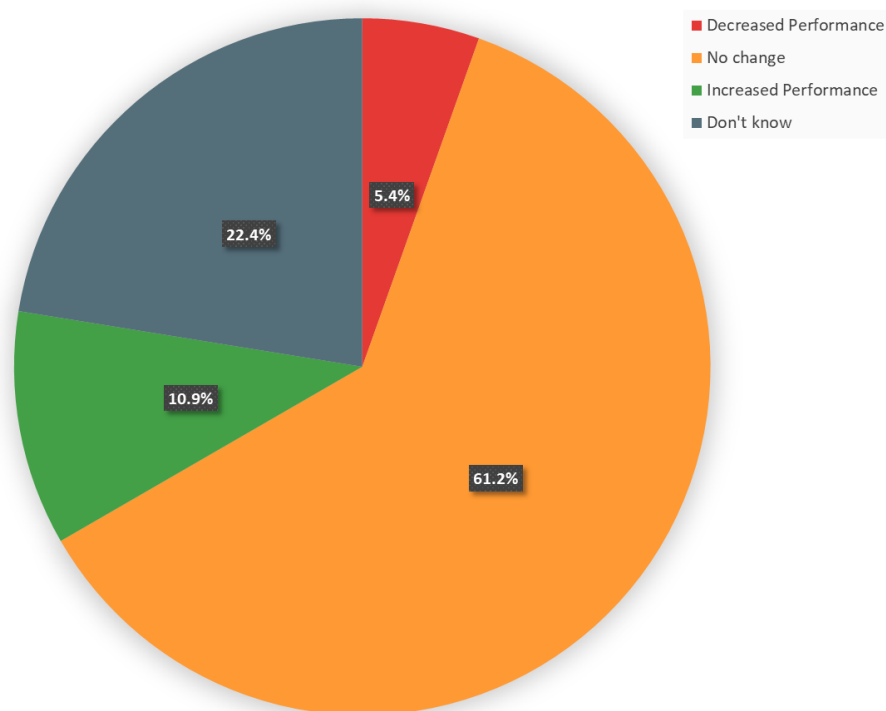
Figure 10: Selections for performance, totalled across all constructs.

4.1.7 Usage In Wider Industry

The final set of questions asked participants about their perception of the prevalence of the constructs in the wider industry of software development, asking whether they thought that use of each construct was above, on par, or below average across the industry in general. Figure 12 collates results across all constructs for this question set.
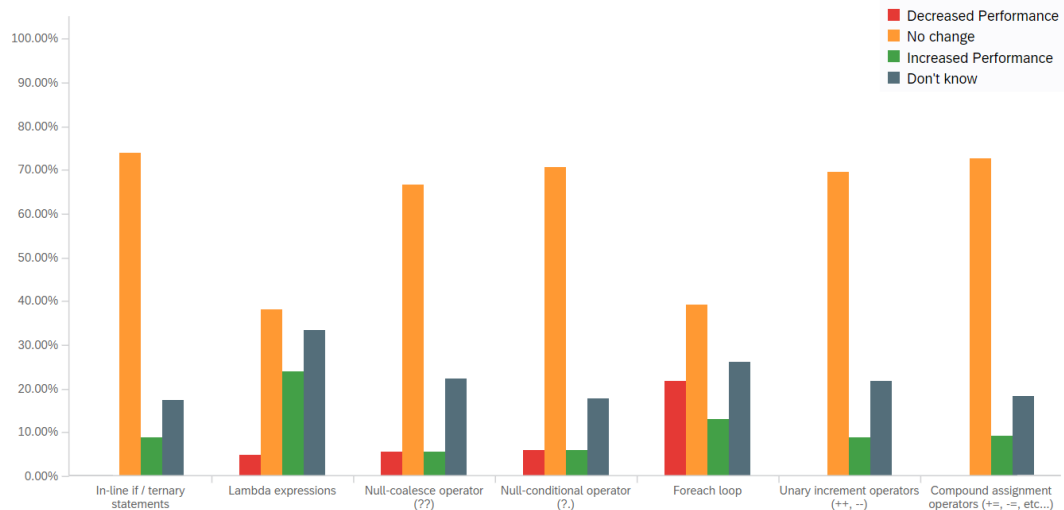
Figure 11: Performance selections grouped by construct. (n.b. Figures normalised to percentages per construct)

In general, responses were quite scattered, and the most clearly discernible patterns are centred on the "Don't know" responses.

The most consistent pattern across the entire set of responses was that of uncertainty around usage of the constructs in Java. This will likely but due to the fact that ISS does not use Java in its tech stack anywhere. The most obvious case of this is for ternary if-statements, with 60.9% of all responses for Java being "Don't know". All four options were roughly the same for C# and JavaScript otherwise. Also, lambda expressions and `foreach` loops were perceived as being more common in C# and JavaScript by a majority.

Aside from this, most results were quite close together, or scatter enough to hide any possible pattern that may be present. Looking at the wider picture, ternary if-statements were the only construct that received a significant number of selections in any language a notable number of "Below average" selections, suggesting that, generally, C#, JavaScript, and Java use ternary if-statements less than other languages in the software development industry.

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

[]

4.2.3 ReSharper

[]

Figure 12: Responses on prevalence in wider industry compared to each language, displayed per construct. A typographical error meant that participants were asked about null-coalesce in Java: this should have been JavaScript, as Java has no null-coalesce operator. That portion of the results should be ignored. (n.b. Figures normalised to percentages and truncated scale, Null-coalesce and null-conditional operators are not present in Java, giving only two groupings)
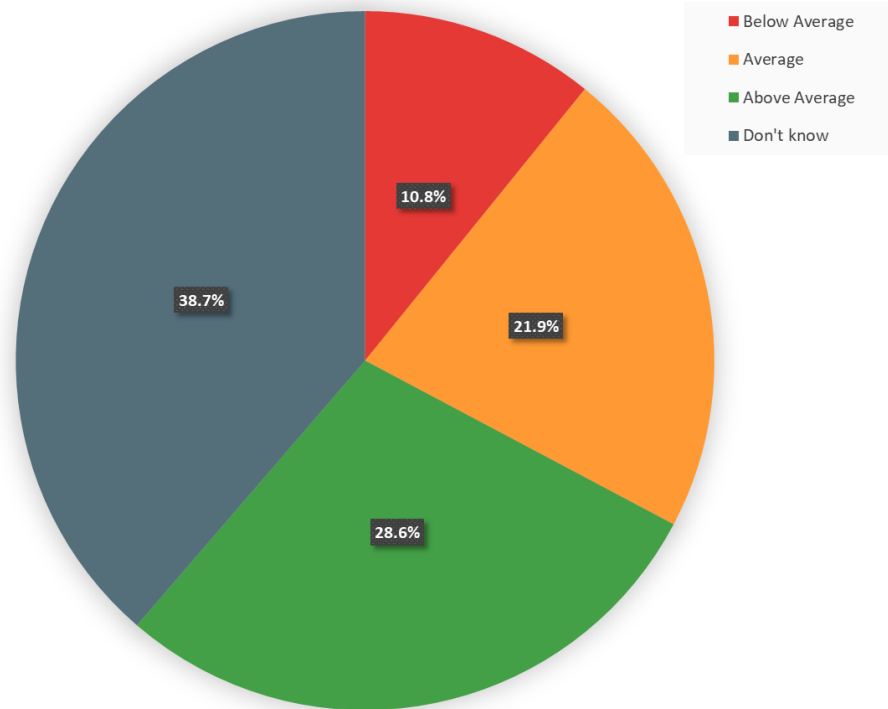
Figure 13: Responses on prevalence in wider industry compared to each language, totalled across all constructs and languages.

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

[]

- survey: what does "complexity" mean? Computational? That'd be covered by slow/faster?
- GitHub repo selection: mistakenly thought it was 'top of all repos for lang., it was not. (Check Tracy emails for reference and remedy, time permitting)'
- Use/Not Use, discrepancy in tense used in presented options, potentially created a bias toward 'use', although this would have bene introduced AFTER the 'use' question (participants can step backward through survey however)

5.2.1 Lambdas

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

Berry, R. E. & Meekings, B. A. (1985), 'A style analysis of c programs', *Commun. ACM* **28**(1), 80–88.
  URL: *https://doi.org/10.1145/2465.2469*

Bloch, J. (2018), *Effective Java*, 3 edn, Addison-Wesley, Boston, MA.

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*

Moghadam, J. B., Choudhury, R. R., Yin, H. & Fox, A. (2015), Autostyle: Toward coding style feedback at scale, *in* 'Proceedings of the Second (2015) ACM Conference on Learning @ Scale', L@S '15, Association for Computing Machinery, New York, NY, USA, p. 261–266.
  URL: *https://doi.org/10.1145/2724660.2728672*

Oman, P. W. & Cook, C. R. (1988), 'A paradigm for programming style research', *SIGPLAN Not.* **23**(12), 69–78.
  URL: *https://doi.org/10.1145/57669.57675*

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

Roy Choudhury, R., Yin, H. & Fox, A. (2016), Scale-driven automatic hint generation for coding style, *in* A. Micarelli, J. Stamper & K. Panourgia, eds, 'Intelligent Tutoring Systems', Springer International Publishing, Cham, pp. 122–132.

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

Various (2021), 'Airbnb javascript style guide'.
  URL: *https://github.com/airbnb/javascript*

Various (n.d.), 'Google javascript style guide'.
  URL: *https://google.github.io/styleguide/jsguide.html*

Appendix

A: Project Proposal

SCC.419/SCC.421 Dissertation Proposal: Examining the Effects of Advanced Programming Language Constructs on the Maintainability and Extensibility of Code

Luca Davies

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

13th March 2021

Abstract

The proposed project is a research-centric undertaking that will examine the usage of a number of optional "advanced" programming language operators and constructs as a lens on the evolvability and maintainability of codebases. The null-coalesce, null-conditional, ternary if, and lambda constructs are of particular interest.

Open source code bases such as those found on GitHub will be employed as example code and will be studied closely. Data will also be gathered via a survey of a small selection of professional developers on their thoughts around these constructs. Together with existing literature and research, these sources will be used as a basis to form recommendations on the best and most appropriate usage of such advanced constructs and how they may be best leveraged to produce code that is highly extensible and maintainable.

1 Introduction

As we press on through the Digital Age, and "ubiquitous computing" becomes a more and more common phrase, we can further assume that, at any given moment, we are within relatively close reach of some form of computing device and its software. More and more aspects of human life are being subject to integration with technology and computing to improve their function, make them more effective, and to *expand* their function. It follows, then, that the code we write to allow this integration should have as much effort put toward it to make the software itself effective and expandable.

All of the code we write should be designed with maintainability and extensibility in mind: it is all but inevitable that the requirements and use-cases of any well-used piece of software will expand as it gains traction, and so as Samoladas et al. (2004) discuss, the ability to grow the codebase to accommodate new features and functions is of great importance. Further, the larger an application is, the more likely it is to be developed not by an individual, but by a team. Writing code, and writing code *well* is a valuable skill - a balancing act in ensuring a program is both functional and clear in its purpose, such that it is not taxing for another developer to read the code and understand how it works.

We can make our code clearer by using the full range of constructs provided to us by our programming languages. Many of these are simple and are introduced to aspiring programmers very early on - others, however, are more complicated and require a little more explanation:

- 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. (**?**)

- 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. (**?**)

On the surface, it might seem like the right choice to always use these constructs: not only can they be used to make certain patterns more compact, but in some cases they can also emphasise the purpose and function of some lines in a much clearer way than more traditional syntax. However, it is not quite that simple: the choice to or to not use them can have adverse effects, both on the readability of our code and, in some cases, the function too. For one, ternary ifs in some languages can prevent (or cause) unwanted side effects from function calls or may be very difficult to read if care is not taken.

Taking all of this into account, this project will explore both existing code and the thoughts of professional developers on these constructs to gain insight into the best usage of them, situations in which they are beneficial, and situations in which they are not.

1.1 A Note on Company Context

An in-depth company context and link to my placement company, Information Systems Services (ISS), is absent from this proposal as there is no strong link to be made. Due to the nature of my placement project, being without much appropriate level of novelty or purely academic merit, there was little scope for a workable master's dissertation. However, this project was still inspired by my placement work, having worked on a significant amount of code, and having learnt some new and more advanced features of C#, primarily the null coalescing and null conditional operators during the process.

Although there is no direct link, I have been offered the freedom to survey and gain insight from many of the developers within ISS, should I wish - an offer I will be making use of for this project.

2   Background

It appears that there has been very little research explicitly into the usage of these language constructs that has been visibly published online. Although this potentially stems from the difficulty to search for the correct terms without interference from irrelevant and unrelated topics, there still seems to be nearly no papers covering this kind of topic. This section will briefly cover the origin of some of the key language constructs that will be covered in this project to provide what background is readily available on the topics.

As programming languages have grown and developed over the last 70 to 80 years, it is reasonable to say that as languages grew more complex and feature-rich, many became extremely verbose, requiring many lines to do even simple calculations. In more recent decades, we have seen this process reverse in some ways with the in the mass adoption of lambda functions and anonymous functions first introduced by ECMAScript in 1997 (Eich 1997), which would later be implemented as JavaScript. This construct allowed programmers to avoid the lengthy definitions normally required for a discreet unit of code. It could be used easily in one-off situations (anonymous functions), in-line and without the overhead of a normal function. Further, these functions may be passed between other functions as arguments (lambda expressions) to allow even more advanced code to be written.

The "ternary if" construct can be traced back to CPL and BCPL (Richards 1967) which would go on to become the B language, which itself would develop in the original version of C, leading to the ternary if being equally ubiquitous in *availability*, if not so much in usage. This can be particularly useful when a very simple if/else structure is required. It allows compression of what might otherwise be 5-10 lines into a single line. Of course, if the if/else statments require evaluating themselves, the expression risks becoming difficult to read.

Lastly, there is no easily available reference for the first appearance of the null-coalesce operator, but it is a newer addition to more modern programming languages, though being present in C# from version 2.0 (2005) onward. Under a different guise, null-coalesce has been in use for quite some time as a means to supplying a default value, but the explicit inclusion of a binary null-coalesce operator remains in the realm of the more recent features in many of the languages used in production code today.

3   The Proposed Project

3.1   Aims & Objectives

The aim of the project is to investigate the usage of the aforementioned advanced language constructs in code, and to further provide recommendations on the best way for the constructs to be employed to ensure maximum ease of understanding within the context of the code and ease of extensibility and editing for future developers.

The constructs to be studied were selected as they are more advanced syntax (that is, they're not generally taught to new developers till a later stage) and have some depth and variation to their usage. There are other potential constructs and operators that could be examined (noted below), but these are only to be examined if time permits. The selected constructs are as follows:

- 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:

  - Unary Increment Operators
  - Compound Assignment Operators
  - Foreach Loops

The end goal will be to provide a set of recommendations for the three primary constructs on how they should be used in code, where code can benefit from them, and where they should be avoided. Failing this, as a minimum, observations will be presented linking usage of these advanced constructs to readability and extensibility as according to the data gathered via survey.

3.2 Methodology

Although the project will not be language-specific, it will be limited to looking at languages that have the specific constructs implemented as part of the core language syntax and are quite popular (e.g. top 10 languages present on GitHub [1]). Thus, C#, Java, and JavaScript will be examined.

The first steps will be to confirm the sources of public code being used to study the above constructs. These would most likely be random or select files from open source GitHub repositories of well-known and popular software and tools. Possible repos to use include those of the .NET Platform [2], CheckStyle [3], and npm [4]. Using basic search tools, a cursory scan of the repos for usage of the constructs to be studied will help define some basic types of use-case for each construct based on a small amount of context each is used in. These use cases will help partition when the use of the construct is actually helpful or not (e.g. a short ternary with concise code, vs. a lengthy ternary with very long code, making it very difficult to read). Finding of the constructs in the repos can be carried out using both normal IDE and code editor text search functions as well as regular expression-based searches (a feature provided by many IDEs and editors). Thankfully, this should not prove too difficult, as the characters that flag these constructs are syntactic and part of the language grammars - they are unique enough to find with ease. Once usages are identified, an ideal scenario would be to examine pull requests and issues to discover if there is any noticeable volume of bugs or issues caused by incorrect use, poor use, or avoidance of use of the advanced constructs that needed to be addressed.

Alongside this, a survey will be created to be sent to a number of developers within ISS to gain both qualitative and quantitative data around use of the constructs within their own work and that of their colleagues. Questions will be asked in the realm of "Do you use <construct> often?", "How easy do you find it to understand using <construct>?", and presenting code snippets both with/without the relevant constructs and asking participants which they find easier to read and/or understand. Statistical methods may also be used to gain better insight into the quantitative data. If time permits, guided interviews may also be held to gain additional qualitative data, assuming ethics clearance that will be submitted before this project commences in earnest.

3.3 Potential Risks

Due to the apparent scarcity of research into this topic, there is a possibility that there will be little variation in the data collected - that is that all evidence points towards a single conclusion without much conflict to discuss or examine. For example, examined code and survey results may show that, in general, developers are writing very good code with these constructs and there's not much that can be suggested for improvement against what has been examined. However, my experience with a number of developers and codebases, as well as the general attitude amongst many developers regarding what might normally be called a "stylistic choice" would suggest that there is likely to be discourse on the topic. One example of this is the existence of many different brace styles for C-like languages, none of which can truly be described as being vastly more common or "better" than others.

Secondly, the open source repositories that are candidate for examination are very large, certainly far too large to hand examine in full. This is why exact files to be examined will be selected at random or for greatest frequency of construct usage. Arguably, this may introduce a bias to the results, as a developer who uses these kinds of constructs often is more likely to have at least consistent usage of them, if not "better" usage too. Although given the size of the repos in question, and the number of collaborators often involved, it seems unlikely that any given file would be totally saturated in the style of one developer.

There remains a reasonable chance of bias from the developer surveys, as all participants will be from the same organisation, and thus may tend to write their code in similar ways. However, they are still different people, in various stages of their careers and learning, and will still have differing opinions and thoughts on the topics this project will examine. In an ideal world, a large number of developers from an array of backgrounds, if not from all over the world, would be surveyed but given the time scale and scope of this project, this is impractical.

---

[1] https://octoverse.github.com/

[2] https://github.com/dotnet, Home of the open source .NET Platform

[3] https://github.com/checkstyle/checkstyle, Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard.

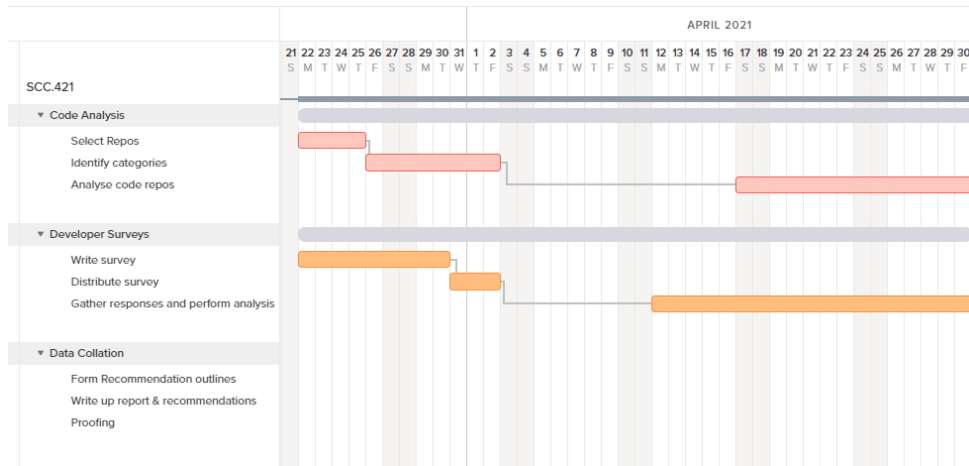[4] https://github.com/npm, node package manager

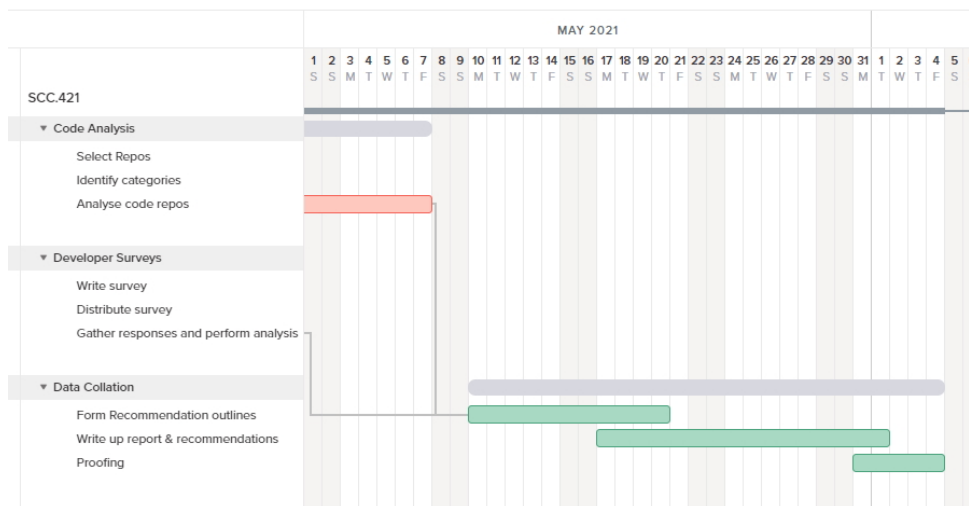*Figure 1: Project schedule, Mar-Apr 2021*

Figure 2: Project schedule, May-Jun 2021

4   Programme of Work

The groundwork for the project will begin in mid-March 2021 and will run till the start of June 2021, with Friday 4th June as a hard submission deadline. As this project is primarily research-focussed and has no specific technical implementation, there are not many well-defined stages of progress, however the following milestones will be used as an approximate gauge:

- Code Analysis
    - Code repositories finalised and acquired
    - Categories for each construct identified and defined
    - Counts of each construct and category found
    - Data collected on coincidence of bugs/issues and advanced constructs
- Developer Surveys
    - Survey written
    - Survey sent to ISS developers
    - Sufficient results from survey acquired
    - Statistical analysis performed on survey results
- Both data sources combined to draw conclusions and make recommendations
- Final project write-up, analysis, reflection, and summary

A potential schedule for the project is show in Figures 1 and 2.

5   Resources Required

No specialist resources are required for this project, only standard tools that are already available to me:

- A code editor or other regex-capable text program (Visual Studio Code)
- Access to survey software (Qualtrics)
- Access to GitHub

References

Eich, B. (1997), *Standard ECMA-262, ECMAScript: A general purpose, cross-platform programming language.*
URL: *https://www.ecma-international.org/publications-and-standards/standards/ecma-262/*

Richards, M. (1967), *BCPL Reference Manual.*
URL: *http://cm.bell-labs.co/who/dmr/bcpl.html*

Samoladas, I., Stamelos, I., Angelis, L. & Oikonomou, A. (2004), 'Open source software development should strive for even greater code maintainability', *Commun. ACM* **47**(10), 83–87.
URL: *https://doi.org/10.1145/1022594.1022598*

B: Survey

Ethics Preamble

The following survey is being conducted by Luca Davies as a student of Lancaster University in pursuit of completion of module SCC.421: Fourth Year Project.
The data you provide will be collected and handled in line with GDPR and will be used for the lawful purpose of research in accordance with safeguards. For more information, please see the university website [here](#).

If you have any question regarding the research itself, please email l.davies7@lancaster.ac.uk.

Preamble

This survey is aimed at gathering the thoughts and opinions of programmers and software developers on certain more "advanced" constructs and patterns available for use in some programming languages such as C#, JavaScript, and Java. A knowledge of all three of these languages is *not* necessary for participation.

The constructs being examined are:

- In-line If / Ternary Statements
  ```
  a ? b : c
  ```
- Lambda Expressions
  ```
  (a) => {...}
  ```
- Null-coalesce and Null-conditional Operators
  ```
  a ?? b
  a?.b
  ```
- Foreach loops
  ```
  foreach ( obj a in b) {...}
  for ( obj a : b) {...}
  ```
- Unary Increment Operators
  ```
  a++
  a--
  ```

- Compound assignment Operators

```
a += 5
a -= 8
etc...
```

Experience

How many years of programming experience do you have?

< 1

1 - 4

5 - 9

10 - 14

15 - 19

> 20

Awareness and Usage

Which of the following constructs are you aware of? Select all that apply.

In-line if / ternary statements

Lambda expressions

Null-coalesce operator (??)

Null-conditional operator (?.)

Foreach loop

Unary increment operators (++, --)

Compound assignment operators (+=, -=, etc...)

I am not familiar with any of these

In places where they can be used, how often do you use each of these constructs in your code?

	Never	Rarely	Sometimes	Frequently	Always
In-line if / ternary statements	O	O	O	O	O
Lambda expressions	O	O	O	O	O
Null-coalesce operator (??)	O	O	O	O	O
Null-conditional operator (?.)	O	O	O	O	O
Foreach loop	O	O	O	O	O
Unary increment operators (++, --)	O	O	O	O	O

	Never	Rarely	Sometimes	Frequently	Always
Compound assignment operators (+=, -=, etc...)	◯	◯	◯	◯	◯

Use

When you choose **to use** these constructs, for what reason do you use them? Select all that apply.

	It is company style	It makes code clearer	It makes code simpler	It makes code run faster	None of these
In-line if / ternary statements	☐	☐	☐	☐	☐
Lambda expressions	☐	☐	☐	☐	☐
Null-coalesce operator (??)	☐	☐	☐	☐	☐
Null-conditional operator (?.)	☐	☐	☐	☐	☐
Foreach loop	☐	☐	☐	☐	☐
Unary increment operators (++, --)	☐	☐	☐	☐	☐
Compound assignment operators (+=, -=, etc...)	☐	☐	☐	☐	☐

You selected "None of these" at least once in the previous question. Please briefly describe the reasons you use the construct(s) for which you selected this option.

[text box]

Not Use

When you choose **to not use** these constructs, for what reasons do you not use them? Select all that apply.

	It is company style	It would make code less clear	It would make code more complex	It would make code run slower	None of these
In-line if / ternary statements	☐	☐	☐	☐	☐
Lambda expressions	☐	☐	☐	☐	☐
Null-coalesce operator (??)	☐	☐	☐	☐	☐
Null-conditional operator (?.)	☐	☐	☐	☐	☐

	It is company style	It would make code less clear	It would make code more complex	It would make code run slower	None of these
Foreach loop	☐	☐	☐	☐	☐
Unary increment operators (++, --)	☐	☐	☐	☐	☐
Compound assignment operators (+=, -=, etc...)	☐	☐	☐	☐	☐

You selected "None of these" at least once in the previous question. Please briefly describe the reasons you choose not to use the construct(s) for which you selected this option.

[   ]

Efficiency

What effect do you think these constructs have on code performance?

	Decreased Performance	No change	Increased Performance	Don't know
In-line if / ternary statements	○	○	○	○
Lambda expressions	○	○	○	○
Null-coalesce operator (??)	○	○	○	○
Null-conditional operator (?.)	○	○	○	○
Foreach loop	○	○	○	○
Unary increment operators (++, --)	○	○	○	○
Compound assignment operators (+=, -=, etc...)	○	○	○	○

Construct Frequency

Considering the wider programming and software development industry, how often do you think **in-line / ternary if statements** are used in these languages?

	Below average	Average	Above average	Don't know
C#	○	○	○	○

	Below average	Average	Above average	Don't know
JavaScript	○	○	○	○
Java	○	○	○	○

Considering the wider programming and software development industry, how frequently do you think **lambda expressions** are used in these languages?

	Below average	Average	Above average	Don't know
C#	○	○	○	○
JavaScript	○	○	○	○
Java	○	○	○	○

Considering the wider programming and software development industry, how frequently do you think **null-coalesce operators** are used in these languages?

	Below average	Average	Above average	Don't know
C#	○	○	○	○
Java	○	○	○	○

Considering the wider programming and software development industry, how often do you think **null-conditional operators** are used in these languages?

	Below average	Average	Above average	Don't know
C#	○	○	○	○
JavaScript	○	○	○	○

Considering the wider programming and software development industry, how often do you think **foreach loops** are used in these languages?

	Below average	Average	Above average	Don't know
C#	○	○	○	○
JavaScript	○	○	○	○
Java	○	○	○	○

Considering the wider programming and software development industry, how often do you think **unary increment operators** are used in these languages?

	Below average	Average	Above average	Don't know
C#	○	○	○	○
JavaScript	○	○	○	○
Java	○	○	○	○

Considering the wider programming and software development industry, how often do you think **compound assignment operators** are used in these languages?

	Below average	Average	Above average	Don't know
C#	○	○	○	○
JavaScript	○	○	○	○
Java	○	○	○	○

Closing

Do you have any further comments on/around the usage of the mentioned constructs?