Comparative analysis of 6 programming languages based on readability, writability and reliability

Ishrat Jahan Ananya 1631636042

Farsihta Jayas Kinjol 1721411042

Zahin Ahmed 1711020042

Sabbir Hasan 1411107042

Bitan Debnath 1721929042

# Abstract:

# Introduction:

A programming language is a formal language used to communicate with machines such as computers. Programming languages consist of a set of instructions, usually called the “syntax”, and rules on how the syntax can be used in combination with each other. The first programming language “Plankalkül” was designed by German scientist Konrad Zuse, however it was never implemented[1]. Short Code, one of the first pseudocode languages, was developed by John Mauchly in 1949. Soon after, FORTRAN became the first high level programming language that was commercially implemented in 1954 by John Backus[1]. Consequently, many languages have been developed over the years for many different purposes. Some languages have been designed to be suitable for complex computations, while others cater to other domains such as Artificial Intelligence. There are two major categories of programming languages: Imperative languages and Declarative languages. Imperative languages such as C, C++, and Java are more commonly used by general people and are often called general purpose languages as well. Declarative languages such as functional languages (LISP) have specific use cases.

While most imperative programming languages have some basic similarities, the difference usually lies in syntax, how the syntax is converted to machine language, efficiency, closeness to machine language (high level vs low level) etc. In recent years, a lot of emphasis has been given on making languages more readable and writable for programmers. Languages such as Python, R, Julia etc. are much higher level compared to the languages they are built on (mostly C, C++, S and Scheme) [2], [3], and they were developed with the aim of higher abstraction and making the whole process of coding easier and more convenient[4]. Indeed, the readability and writability of code holds great importance to all types of users, starting from non-programmers, novice programmers or even experienced programmers. In today’s world, almost everyone has to deal with some type of programming or code. People with no experience in coding will not be comfortable in reading or understanding code that is complicated or unable to express its purpose clearly. Similarly, novice programmers will face a huge learning curve if the syntax and rules of a language are not orthogonal and simple[5]. In this case both readability and writability are important, as they are trying to learn the syntax while also implement it themselves. Finally, experienced programmers usually deal with code that is implemented and used in industry. Such codebases are maintained over long periods of time, and the programmer(s) dealing with the code changes often. Hence, not only are readability and writability extremely important here (for the programmers to be able to understand and extend/modify the code as needed), but also reliability comes into play. Reliability of a programming language ensures that the code behaves the way it is supposed to, and carries out its purpose, at all times. This is very important in industry-level programming where the smallest of unforeseen circumstances can cause huge losses.

Hence, the aim of this research is to compare the readability, writability and reliability of 5 commonly used imperative programming languages, which are C, C++, Java, JavaScript, Python, and 1 functional language, which is R. All of these programming languages can, and is currently being, used for similar applications hence it is important to compare the differences in readability, writability and reliability of the languages, and the trade-offs that accompany the differences. The study consists of a theoretical comparison between the constructs and design of the programming languages to judge readability, writability and reliability, a survey conducted using code snippets to judge just readability and writability. Finally the readability and writability were compared to the runtimes of two algorithms to analyze whether there’s a trade-off between efficiency and these two factors, and we also tried to check for reliability issues while running these computationally intensive algorithms.

# Literature Review

# Readability, Writability and Reliability

Before embarking on the journey of comparing these three metrics for each language, it is important to understand how each is defined and what they signify.

## Readability

As the term suggests, “readability” of a language is essentially how easy it is to read a piece of code and understand what it is doing [1]. When earlier programming languages were developed, the main factor to be considered was efficiency of the language. However, as the use of these languages increased, it became apparent that if users were taking a long time understanding an existing piece of code, then it was slowing down the process and hence reducing efficiency before the computation even began. Hence, eventually the focus shifted from machine oriented efficiency to human efficiency. Depending on the problem domain, the concept of readability changes. For example, a language that is not meant for complex numerical computations, such as vanilla JavaScript, might not make a very readable script for mathematical computations.

## Writability

The writability of a programming language is defined by the ease of creating a new program for a specific problem domain [1]. Similar to readability, writability of a language is also heavily influenced by the problem domain it is being used in. An easy and common example for this would be the difference in writability for a program with a Graphical User Interface (GUI). A language such as Visual BASIC or Java would have high writability for such programs as they are designed for such applications, whereas C would have very low writability as it is simply not designed for programs that require a GUI.

## Reliability

The performance of a program depends considerably on the language it has been written on. Its affects the decision of whether the program will be reliable for further extension and modification in the future if required. Of course reliability is a more abstract term and not all languages need to have the same standard and it depends on the purpose of the program it’s written with.

A significant criterion to base our judgment has been mentioned in John D Gannon and J.J Horning’s paper “Language Design for Programming Reliability*”*[6] which is the ability of a programming language to decrease programming errors and the ability to detect them if any. This can be related to the readability and writability of the said programming language. A language that is more readable theoretically should have less fuss to deal with when trying to detect anomalies or errors.

In the table below, factors that affect readability, writability and reliability are shown. Short descriptions of each factor is given afterwards.

Table 1 Factors affecting Readability, Writability and Reliability

|  |  |  |
| --- | --- | --- |
| Readability | Writability | Reliability |
| Simplicity | Simplicity | Simplicity |
| Orthogonality | Orthogonality | Orthogonality |
| Data types | Data types | Data types |
| Syntax design | Syntax design | Syntax design |
| Comment style | Variable naming conventions | Support for abstraction |
| Indentation/White spacing | Support for abstraction | Expressivity |
| Variable naming conventions | Expressivity | Type checking |
|  |  | Exception handling |
|  |  | Restricted aliasing |

## Factors

* Simplicity:

Simplicity of a language is exactly what the term suggests, and there are many factors that affect simplicity of a language. Firstly, the complexity and size of the language constructs of a programming language affect its simplicity. Languages that have a large number of basic syntaxes are often difficult to learn, and often they are not fully mastered by a programmer. This poses as a problem when a reader faces constructs they are not familiar with and finds the code unreadable, even though they might be an expert with a different subset of the language [1].

Another issue that reduces simplicity of a language is feature multiplicity. When the same task can be carried out in multiple ways using different syntax, the language is said to have high feature multiplicity; this makes the language more complex as someone who is habituated to using one, will have a hard time understanding the other. [1]

Operator overloading also increases the complexity of a language. While it is usually seen as an increase in flexibility, it also creates the possibility for users to create operations that do not align with the function’s original meaning or is distinctly different from it. For example, a user might define a function subtract using the “+”, and that is possible through operator overloading. Hence this will greatly confuse the reader or future contributor, immensely reducing the readability.[1]

* Orthogonality:

The orthogonality of a language can be defined in two ways. One, it is the concept of one operation doing only one task at a time and not affecting any other variable or condition [7], in other words, there is no side effect of the operation and there is only one way to do the operation. Another definition is that, all language constructs are independent of each other, hence they can be logically combined together to control and make data structures as necessary [1]. If the constructs are independent of each other, they will not behave differently in different contexts. Hence, languages that have context-dependent syntax are not orthogonal. Moreover, languages that have many exceptions in rules, such as return types or parameter passing, are not orthogonal.

* Data types:

There should sufficient numbers of data types present in the language such that all types and sizes of data can be accurately represented and stored in the memory [1]. Some languages do not have a primitive data type for string or Boolean, and this often reduces the readability and writability of the language as these variables have to be expressed as something they are not.

* Syntax design:

Syntax is essentially the building blocks of a language. Logical syntax that reflect their purpose clearly is an obvious aid to readability. Another point to consider is the way compound statements are designed. In many languages, compound statements or statement groups are not easily readable as it is hard to judge the hierarchies of the statements, and where they began/end.

* Comment style:

Comments make a code readable as they usually describe what is happening in the code and its purpose. However, how these comments are structured affect reliability as well. There are mainly 2 types of comments: single line comment and multi-line comment/ block comment. Among programmers, it is general consensus that single line comments are more readable than block comment, as it often becomes difficult to understand where the comment stops and code starts again, especially if someone is using a non-color coded environment [5].

* Indentation/White spacing:

White spacing, or gaps in between lines of code, are treated differently in different languages. In some languages, white spaces are simply disregarded and no indentation is needed for the code to function properly. However, indented code is far more readable than non-indented code, as the indentations make the hierarchy of compound statements much more apparent. Therefore, languages that enforce indentation for loops or conditional statements, have much better readability. Languages that enforce every new instruction to be on a new line are also more readable.

* Variable naming conventions:

Conventions enforced by a programming language can be seen as both increasing/decreasing the readability. This is highly subjective, as someone who is familiar with the conventions will find the code much more readable when the conventions are followed, whereas someone who is accustomed to another convention might find the same code strange. However, it is general consensus that having some set conventions is better as it encourages the programmer to follow better programming practices.

* Support for abstraction:

When it comes to readability and writability, support for abstraction is paramount. Using classes to encapsulate elements of an object, using functions to declare procedures only once and use them multiple times, all contribute to making the code readable and cluster-free.

* Expressivity:

Expressivity refers to the power of a language by which it can express complex procedures using short notation or fewer lines of code. This in turn leads to a bigger number of constructs in the code, some of which are quite powerful and hence might reduce the overall simplicity of the code. However, fewer lines of code usually reduce the time needed for a reader to understand it [1].

* Type checking:

All languages check for type errors now, but the main question is when this checking is done. Run-time type checking reduces the efficiency of the program greatly, hence compile-time type checking is more reliable and desirable. Moreover, the earlier errors are detected (usually by compiler), the easier and less expensive it is to fix the error.

* Exception Handling:

Regardless of the number of checks done in compilation, there will always be unforeseen circumstances in runtime. What facilities a language provides in dealing with these unpredictable or predictable unwanted circumstances is important, as otherwise programmers cannot make a stable program that runs reliably or gives reproducible results.

* Restricted aliasing:

Aliasing is when two pointers are directing to the same memory location. This is undesirable as we don’t want mismatched data, undefined results or overwrite any value. Thus a strict aliasing rule is often integrated within the language system to prevent pointers of different objects to never indicate to the same memory location.

The above factors, and what should/ should not be their values for a readable, writable and reliable language is summarized in the table below:

Table 2 Metrics to define readability, writability and reliability of languages

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Simplicity | Orthogonality | Data type | Syntax design | Comment style | Whitespace /indentation | Variable naming conventions | Support of abstraction | Expressivity | Type checking | Exception handling | Restricted aliasing |
| Number of constructs  (lesser the better) | Context dependent syntax?  (should not be) | Support available for all data necessity?  (should be available) | Form and meaning  (should be related to the meaning) | Allows single line?  (should allow) | Indented?  (should be) | Enforced by language?  (should be) | Level of support  (should be high level support) | Average number of lines needed to write a program  (should be fewer) | Done in compile time or run-time?  (should be compile time/ statically) | Available?  (should be available) | Provided?  (should be provided) |
| Feature multiplicity  (should not be supported) | Return types  (should be consistent) |  | Compound statements  (should have definitive ways to signify hierarchy) | Allows multi line?  (should not allow) | White spacing disregarded?  (should not be) |  |  |  | Strongly typed?  (should be) |  |  |
| Operator overloading  (should not be supported) | Exceptions in rules(return types, parameter passing)  (should be consistent) |  |  |  |  |  |  |  |  |  |  |

# Programming languages

As mentioned earlier, the six programming languages used in this research are C, C++, Java, JavaScript, Python and R. In this section, some basic information about the languages are given, to provide some context about certain features they may or may not have.

### C

C is one of the earliest and most impactful programming languages in history. It was developed in the early 1970s at the Bell Laboratory by American computer scientist Dennis M. Ritchie and Ken Thompson. It was initially developed to be used in UNIX operating system. When C was introduced it had a great combination of features as it was simple, portable, structured and extensible. It also had a richer library and better memory management system with pointer and recursion ability than other languages at that time.

### C++

C++ was created by Bjarne Stroustrup as an extension of C programing language that introduced class functionality, which is why it was initially called C with classes. It was renamed C++ in 1983. The language started to expand its features with classes, basic inheritance, function argument, virtual functions etc.. Without compromising speed and portability, C++ has kept expanding its feature set to stay relevant decade after decade.

### Java

A small team of engineers called Green Team in Sun microsystems initiated the JAVA language project in 1991. It was initially developed for interactive television like set top box, but was very advance for cable television back then. Java is robust, secured, platform independent, high performance object-oriented programming language and it is reasonably fast. Java provides a software based platform. Java virtual machine is what make java so portable. Java program is compiled to bytecode and then executed by virtual machine that is specifically designed for the host hardware.

### JavaScript

JavaScript, also known as JS, was created by Brendan Eich at Netscape. It was officially called LiveScript when it was first became a part of Navigator release in 1995. Later it was changed to JavaScript probably as a marketing ploy as Java was then a hot new language. JavaScript is one of the core technology of World Wide Web. It enables interactions in website. It is a multi-paradigm language that supports event-driven, functional programming style.

### Python

Guido van Rossum started working on Python in 1989 as a successor to ABC programming language. It was first released in 1991 although, version 1.0 of Python was released in 1994. Although it is not as fast as other popular programming languages such as C/C++ or Java, its main point of focus is ease of learning, alongside short and easy-to-code syntax which made it one of the most readable programming languages. It is now widely used in technical fields like machine learning, artificial intelligence and others.

### R

R is a programming language that is specialized for statistical computing, developed by two statisticians Ross Ihaka and Robert Gentleman in early 1990. R implements various statistical and graphical techniques, including linear and nonlinear modeling. It is not very fast for computationally intensive tasks, however it can link with C, C++ and other languages to implement those tasks. R has a strong graphical capability that is why it is vastly used for data visualization. R has a highly active community who contribute in expanding the language’s code repository CRAN, hence the use of R in data science project is increasing.

# Methodology

For the purpose of our paper, we defined a unique procedure to compare the six languages based on readability, writability and reliability. First of all, we created an evaluation metric of three standards : ‘Bad’ , ‘Moderate’, ‘Good’ to judge each of the languages for every criterion in table 1. This is done to form a theoretical comparison based on the characteristics of each language. However, since the evaluation metrics are quite abstract, and very subjective, we chose to conduct a survey to check whether our theoretical comparison is reflected in the preference and judgement of actual users. Hence, we detailed a set of questions in the form of a survey which is further discussed in section 7b. Finally, we ran experiments to compare the runtimes of algorithms written from scratch in these 6 languages, to check how the readability and writability of a language affects its efficiency, if at all. Our assumption was that languages with high readability and writability are closer to the English language i.e.: higher level, hence they should be slower than comparatively lower level/ less readable languages.

To visually explain the findings of our secondary research, we produced the table below that concisely mentions the performance of each of the languages.

Judgement criteria used:

|  |  |  |
| --- | --- | --- |
| **BAD** | **MODERATE** | **GOOD** |

For readability:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Simplicity | Orthogonality | Data types | Syntax design | Comment style | Whitespace/ indentation | Variable naming conventions |
| C | Moderate number of constructs | Some constructs are context dependent [5] | Multiple data types available just for integer type, but no primitive type for string | Form and meaning reflects purpose | Allows single line comment | Not indented | Enforced by language |
| Supports feature multiplicity[5] | Return types not consistent [5] | Compound statements signified by curly braces | Allows multi line comment | Whitespace disregarded |
| Does not support operator overloading | Has exceptions in rules [5] |  |
| C++ | Large number of constructs | Some constructs are context dependent [5] | Sufficient number of data types available | Some syntax do not reflect their meaning (such cout, cin) | Allows single line comment | Not indented | Enforced by language |
| Supports feature multiplicity [5] | Return types not consistent [5] | Compound statements signified by curly braces | Allows multi line comment | Whitespace disregarded |
| Supports operator overloading | Has exceptions in rules [5] |
| Java | Large number of constructs | constructs are not context dependent [5] | No support for complex numbers | Form and meaning reflects purpose | Allows single line comment | Not indented | Enforced by language |
| High feature multiplicity[1] | Return types consistent[5] | Compound statements signified by curly braces | Allows multi line comment | Whitespace disregarded |
| Supports operator overloading | No exceptions in rules[5] |
| JavaScript | Large number of constructs | Commonly used constructs are not context dependent | No support for complex numbers | Some syntax are a bit vague | Allows single line comment | Not indented | Enforced by language |
| High feature multiplicity | Return types consistent | Compound statements signified by curly braces | Allows multi line comment | Whitespace disregarded |
| Does not support operator overloading | Few exceptions in rules |
| Python | Fewer number of basic constructs | constructs are not context dependent [5] | Everything is an object, proper support available for data types | Form and meaning reflects purpose, except list and tuples | Allows single line comment | indented | Enforced by language |
| Mostly no feature multiplicity [5] | Return types consistent[5] | Compound statements signified by indented blocks | Does not allow multi line comment | Whitespace not disregarded |
| Supports operator overloading | No exceptions in rules[5] |
| R | Fewer number of basic constructs | constructs are not context dependent | Everything is an object, proper support available for data types | Form and meaning reflects purpose, but some are quite different than common languages | Allows single line comment | indented | Enforced by language |
| No feature multiplicity | Return types consistent | Compound statements signified by indented blocks | Does not allow multi line comment | Whitespace not disregarded |
| Supports operator overloading | No exceptions in rules |

For writability:

|  |  |  |
| --- | --- | --- |
|  | Support for abstraction | Expressivity |
| C | No support | About 5 lines needed to print hello world |
| C++ | High level abstraction | About 6 lines needed to print hello world |
| Java | High level abstraction | About 5 lines needed to print hello world |
| Python | High level abstraction | 1 line to print hello world |
| JavaScript | No built in support | 1 line to print hello world |
| R | High level abstraction | 1 line to print hello world |

For Reliability:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Type checking | Exception handling | Restricted aliasing |
| C | Static type checking | Does not provide support | Support provided |
| Not very strongly typed[5] |
| C++ | Static type checking | Moderate support [5] | Support provided |
| Mostly strongly typed[5] |
| Java | Static type checking | Strong support using catch and throws | Handles aliasing in runtime, so moderate support |
| Mostly strongly typed[5] |
| JavaScript | Dynamic type checking | Moderate support using try catch throw | Aliasing not possible |
| Not strongly typed |
| Python | Dynamic type checking | Strong support using try except[5] | Support not provided |
| Mostly strongly typed[5] |
| R | Dynamic type checking | Moderate support | Aliasing not possible |
| Strongly typed |

Based on our findings so far, Python and R seem to be the most readable and writable. Both of these languages have become very prominent languages in recent years, which might be attributed to their high level readability and writability. C has very low readability and writability, which makes sense considering it is the oldest and lowest level language amongst these 6 languages.

However, in terms of reliability, Java seems to be the reigning champion. Java has been continuously developed for many years, and is one of the main languages used in industry level development and software. Hence it does make sense that it has the most reliable characteristics. JavaScript and R both have low reliability, which is consistent with the fact that JavaScript is mostly used in web and server applications, but not other types of software and R is not used for software development at all.