STM JOURNALS

Recent Trends in
Programming Languages

ISSN: 2455-1821
Volume 11, Issue 3, 2024
September–December
DOI (Journal): 10.37591/RTPL

https://journals.stmjournals.com/rtpl

Research RTPL

Trends in Computer Programming and Language

V. Basil Hans[1,*]

Abstract

The field of computer programming has undergone significant transformations driven by evolving technologies, increasing demand for more efficient software solutions, and the continuous need for enhanced system performance. This paper examines the latest trends in programming languages and methodologies that are influencing the future of software development. Key trends include the rise of functional programming paradigms, the growing importance of concurrent and parallel programming to handle multi-core processors, and the shift towards domain-specific languages (DSLs) to meet specialized needs. Additionally, the increasing emphasis on software security has led to the development of languages and frameworks that prioritize safety and vulnerability prevention, such as Rust and TypeScript. Low-code and no-code platforms are making programming more accessible, allowing individuals without coding skills to develop their software solutions. The growing influence of artificial intelligence in automating code generation, testing, and optimization also represents a paradigm shift. This paper provides an overview of these trends, offering insights into how they are reshaping modern programming practices and influencing the future of software engineering.

Keywords: Computer programming, artificial intelligence, Java, Rust, Kotlin, machine learning

INTRODUCTION

The rise in digital literacy has led to a highly skilled workforce, enabling India to lead in fields such as software development, data analytics, and emerging technologies such as artificial intelligence (AI). However, despite the extensive use of digital technology, challenges remain, including the digital divide, especially in rural areas; cybersecurity threats; and the need for upskilling in an increasingly automated job market [1].

Computer programming is a rapidly evolving field. What were once considered breakthroughs in programming today might appear too elementary owing to rapid advancements in technologies such as the Internet of Things, big data, AI, and machine learning, among others. The success of any information processing application now depends largely on the programmer's skill. Fields as diverse as science, engineering, education, entertainment, and medicine have been dramatically changed by technology and will continue to be changed by variations and extensions of existing technologies. In addition, new types of programming tools and languages have enabled programmers to write more robust and complex software applications. This essay presents the most recent trends in computer programming paradigms and languages by covering six areas: (1) the top programming paradigms being used in the software industry today; (2) several programming languages that are being developed as cleaner and faster versions of existing languages or are seen as testbeds for various new language techniques; and (3) the ways programming is evolving through the use of AI and neural networks. Additionally, he will look at what the future of programming languages might hold based on what he sees as the direction in the industry today, some ideas from research, and historical trends [2].

***Author for Correspondence**
V. Basil Hans
E-mail: vhans2011@gmail.com

[1]Research Professor, Department of Management and Commerce, Srinivas University, Mangaluru, Karnataka, India

Received Date: October 26, 2024
Accepted Date: October 28, 2024
Published Date: November 05, 2024

Citation: V. Basil Hans. Trends in Computer Programming and Language. Recent Trends in Programming Languages. 2024; 11(3): 28–35p.

© STM Journals 2024. All Rights Reserved

PROGRAMMING PARADIGMS

Programming paradigms are general methods of software development, each with its strengths and weaknesses. The procedural programming language was the first programming paradigm developed. C and Pascal were the first procedural programming languages that emphasized procedure and function. The procedure and function take some input, perform tasks, and provide output. In this paradigm, the data and methods are separate entities within a program. The procedural or structured programming paradigm adopts a structured top-down approach. In structured programming, modularization is used to divide the program into smaller, understandable, manageable, and reusable parts called functions [3]. It does not allow data to be freely accessed from all parts of the program. Following procedural programming, an object-oriented programming paradigm (OOP) was developed. SmallTalk was the first OOP programming language. In OOP, implementing the functionality of the program using classes and objects brings the object closer to programming. With OOP, a class is a collection of both the data and methods that operate on the data. An object is an instance of class. The main concepts around which the OOP is designed are encapsulation, inheritance, and polymorphism. These concepts enhance code functionality and offer reusability. C++ and Java introduced the concepts of encapsulation, inheritance, and polymorphism to provide more security and freedom to programmers to maintain and provide flexibility to the code. After the OOP, a functional programming paradigm was developed. The focus of this paradigm is a function. In functional programming, these functions are first-class citizens. A function is considered a value, is allowed to pass through an argument, and returns a single output every time. The functional programming paradigm focuses on the values returned by the functions. All functions in functional programming must adhere to two important rules: pure function and immutability [4].

Procedural Programming

Text, a versatile language for computer programming, has evolved significantly in recent decades. This modern development began in the early 1970s and has not yet reached the limits of what is possible. In the 1970s, BASIC was first developed. It had a better command structure, English use, and so on. Then came ALGOL. In the late 1970s, after years of development, new trends in language were developed, moving away from the sequence of commands in a fixed order to OOP [5].

Procedural programming marks a time in the history of computer programming when a flat linear sequence of instructions is used. Software or applications are written in such a manner that they are viewed structurally as a sequence of tasks or procedures to be undertaken before completion. The focus is on using procedures to execute or perform a given task or a set of tasks. The concept of control features plays a significant role and is supported. The control structure is a feature that allows the program to make decisions, that is, either to select between making a yes or no decision or to repeat a series of steps executed by the program's true/false options or performing one task or the other until the condition for stopping this task is met. The procedural programming method for writing computer programs is the simplest and clearest. All novice programmers choose to start learning programming from this type with the rigid belief that it will fetch quick results and understanding. The C programming language is one of the best examples of a procedural programming language. Other examples of such languages include Pascal, COBOL, ALGOL 68, and Pop, among many others [6].

Object-Oriented Programming

Perhaps the biggest fundamental shift in programming came from the introduction of OOP. This change was a paradigm shift, meaning that it represented a new philosophy and approach to programming. Prior to OOP, complex software solutions were created using a process known as procedural programming. Programs written in this way revolved around procedures or functions. OOP, on the other hand, focuses on objects as the building blocks of the software.

In OOP, objects are designed to model entities in the world, both real and conceptual. This concept involves developing software in terms of real-world objects and having objects interact with one another. The main components of the OOP are inheritance, polymorphism, and encapsulation. Encapsulation

© STM Journals 2024. All Rights Reserved

involves grouping data and operations that manipulate the data into a single unit called a class. Inheritance defines the relationship between classes and allows one class to inherit from another class. This enables new classes to reuse and maintain the functionality of existing classes. Polymorphism allows the reuse of a single reference to access different types of objects [7].

One of the primary goals of OOP is to encourage the reuse of code. Creating reusable classes allows for more modular and organized code and makes multiprogrammer projects more manageable. Real-world implementations of OOP are commonly found in video games, which use objects such as a Zombie object, player object, item object, and User Interface objects to represent entities within the game. Professional software projects rely heavily on OOP to create sustainable software. Two of the most popular languages that use the OOP are Java and C++. Java was initially designed to be more efficient than C++, an object-oriented language. The employment of the OOP methodology allows large projects to be broken down into subsections and each piece to be farmed out to a group of developers, each of whom might be working on a different part of the project and not even need to know how large components are created, thus saving time in the implementation process [8].

Although there are many advanced aspects of working with objects in Java, even a basic program can demonstrate concepts simply and elegantly. Consider a Java program with two objects: a circle and a rectangle. The main program might be interested only in drawing the circle and rectangle, but each object also requires both location and size to be defined. Even though the main program ignores the complexities involved in dealing with locations and sizes, the objects can carry this data around inside them and deal with it in their logic. In effect, the main program can simply pass a circle to be drawn, regardless of the location or radius of the circle or the procedure for drawing the circle. Hence, the use of OOP principles helps standardize the operation of entities on the computer and makes the software developer's job far simpler. For these reasons, OOP has become an extremely popular programming philosophy [9].

Functional Programming

Functional programming (FP) is one of the most interesting and growing paradigms in software development. The core of functional programming is the heavy use of pure functions and immutability, which means no side effects or management of states. This is the main difference from classical languages, which are mainly procedural and object-oriented. Some of the main concepts of functional programming are first-class functions, which means functions can be used and treated as any other variable, and higher-order functions, which are the ability to pass one function as an argument or return it as a value. Other interesting concepts are the map, filter, reduce, and fold operations, which are commonly used to iterate through an array or a collection, recursion, currying, and many others. Although almost all existing programming languages have features related to functional programming, few are purely functional [10].

The main functional programming languages, or those that are inspired a lot by functional programming, are Haskell, Scala, Erlang, Clojure, OCaml, F#, Python, R, Elm, Julia, and many others. Haskell is the purest functional language, but it is also the most difficult to learn because it is syntactically different from other languages. It is also a lazy language. Scala is one of the youngest functional programming languages and runs on the Java Virtual Machine (JVM). Scala has an object-oriented concept as well as functional programming. Concurnas, Nim, Crystal, Rachel, Hack, and Idris are newly developed languages that are marked as functional programming languages as well. The major application of functional programming languages, especially in the contexts of Haskell, Erlang, OCaml, F#, J, and Julia, is handling large-scale applications for data processing and concurrent processing. Functional language features maintain good scalability, fault tolerance, parallelism, control of side effects, testability, and maintainability of large software systems.

EMERGING PROGRAMMING LANGUAGES

Rust is a system programming language known for its memory safety and built-in concurrent features. It is designed for use in scenarios where predictable performance and reliable program behavior are

© STM Journals 2024. All Rights Reserved

important. Rust can be used for both server-side and client-side software development. It is particularly favored for creating system-level software, such as file systems, operating systems, and game engines. Kotlin is designed to be fully interoperable with Java and has been officially adopted as an Android development language. Kotlin can compile its code to JVM bytecode, JavaScript, and native code. Many developers say that Kotlin can increase developer productivity. By building Kotlin, developers can avoid creating some things manually. Kotlin's improved type inference system cuts out a lot of redundant code, leaving developers with a more streamlined codebase. Swift went from an Apple-only language to an open-source language and one that could be used to create cross-platform functionality for mobile development, with many developers considering Swift for Android apps. Swift was created as a type-safe and memory-safe alternative to Objective C. In addition to emphasizing safety, Swift offers modern features, and its less verbose syntax makes it easier to use. With heavy optimization features, Swift can save memory and accelerate app performance, making Swift useful for developing apps for iOS, watchOS, and tvOS. Emerging languages are typically designed to solve problems that other languages have failed to solve. Part of this struggle involves writing safe and performant software. These emerging languages aim to be logically safe. Many of them are low-level languages adapted to 21st-century developers; today's developers expect real-time graphical user interfaces, drag and drop, and other familiar, easy-to-use interfaces that can manage the underlying code automatically. Emerging languages are part of a larger trend towards efficiency and performance. Their rise in interest signals a new conversation about speed and improved developer productivity. Further, increasing the likelihood of an emerging language's success is its capacity to fit into an existing ecosystem or pair well with a reliable framework.

Rust

Rust is a system programming language created by Mozilla that was designed to be a safe form of C++. Instead of explicitly having an unsafe mode, Rust makes the default implementation safe, so that unsafe people do not risk others' memory space. Rust's safety and performance features make it a good choice for system programming, game development, or scientific computing, and it is the primary language for the Redox operating system. Rust safety is enforced through its ownership model and, more visibly, borrow checking via its scoped reference types. These checks ensure that programs are memory-safe, and data races are outside the domain of possible bugs that a programmer might accidentally commit. This makes flawless multi-threaded support possible since no destructors race with other threads, which can lead to use-after-free or double-free errors because the destructors are newly being run in multiple threads concurrently.

Rust sports are a strong-type system, with type inference allowing it to have mainly implicit typing and modern expressive syntax features such as garbage collection, pattern matching, and similar features to functional languages. Rust, stable as of 2018, has a smaller set of libraries and external services than languages such as Python, but new services such as WebAssembly, major cloud computing providers, and game development have attracted serious interest. Cargo is Rust's dependency management service and its package manager, similar to other package managers. The community, such as C++ or Java, also has a large presence and a portion of resources devoted to administering the release and alpha testing of the Rust compiler and surrounding tools. Although Rust is a new system programming language, elements of syntax and design are similar to C++ and other existing systems programming languages. Swift developers have historically found it easy to write Rust and vice versa, partly because they were interested in rewriting Swift to Rust. Others have reported a longer learning curve when switching from other languages to Rust; it is difficult to quantify those reports at this time.

Kotlin

Introduction

The broadly unfolded programming landscape encompasses a range of programming languages. Kotlin, a novel programming language, endeavors to foster developer productivity through expressive syntax and natural solutions. Kotlin is distinguished by its compatibility with the established Java programming language. To this end, Kotlin runs on the Java Virtual Machine, thereby ensuring full

© STM Journals 2024. All Rights Reserved

access to Java libraries and leveraging the advancement of the Java platform with support for newer constructs such as coroutines, one of the unique features introduced by Kotlin in the Android ecosystem.

Language Overview

Kotlin is a modern programming language that focuses on concise, expressive, and interoperable. Its syntax is very clear and legible, helping developers write error-free software faster and with fewer boilerplates. It places a strong emphasis on close alignment with Java, focusing on the ability to interoperate with Java codebases. However, Kotlin provides relief from many Java pain points, including null safety, extension functions, and a class of constructs to support data and enumerations that are first-class citizens, rather than a heavy-object construct. The compact syntax and support for null safety dwarf its other common counterpart, Java. These constructs and traits curtail the need to write additional testing suits, as the code has been written and designed to be more deterministic. Along with its use for cloud-native computing, Kotlin, a niche yet very revolutionary, use case is in web and mobile app development, primarily in the mobile app domain where Kotlin's dominance can be noted. In the mobile app industry, over 60% of Android developers have been accepted and moved into the Android space as the first choice of language for Android app development. In the mobile app domain, Kotlin has been primarily chosen for its power, light footprint, and interoperability with Java and other languages, where the majority of transformations are made by both developers and build tools. It boasts features, such as data classes, and leverages a large ecosystem to abstract the majority of boilerplate codes, resulting in a markedly simplified developer experience. Moreover, it has a rich and active developer community, and only code examples, packages, and tools are clicked away. Therefore, it is arguably one of the easiest languages to learn and put together a quick proof of concept. Kotlin enhancements and emphasis on SQL support provide unique extensibility and modularization to enable the interlanguage extensibility commonly used to develop mobile apps and games. In the e-commerce field, it has shown good use in some case studies with advancements in e-commerce, market generation process development, and game development. Kotlin has evolved to support modern programming paradigms and trends and has supplanted Java as the first language for Android development, reflecting its paramount position in non-iOS mobile app development. The broad range of resources and language features, ranging from Android and iOS development to noise eradication, type safety, and extension functions, enables Kotlin programming to remain viable in an open-source environment and the newer mobile app development paradigm. Kotlin's development also runs the risk of losing support if problems occur, such as if the app is built on significant kernel services, but Kotlin's Java interoperability, its active users, tutorials, and guides greatly reduce this risk.

Swift

Introduced in 2014, Swift was designed to be used with Apple's programming systems, iOS and macOS, and is exclusive to Apple platforms. This programming language emphasizes both safety and speed in programming and is intended to replace Objective C, which has been a mainstay for the past 30 years. After six years of active development, Swift developed a modern syntax that was designed by incorporating a wide range of other languages, aiming for all developers to read the syntax as needed. Its strong inference system and safety features make Swift applications less prone to instability, resulting in more robust applications with less strict high-system mechanisms. Swift includes key features to improve software performance, such as options that allow the creation of clean, human-readable code that is safe for null values, type inference for making code safer and easier to read, and protocol-oriented programming that enables writing syntax or designing reusable code logic. These features provide Swift advantages in terms of compatibility, speed, and resilience in the development of mobile applications. Even in terms of development, there are many forums and communities as well as tools to learn Swift more easily and widely.

Swift is designed with safety in mind and empowers developers, including those without a computer science background, with the freedom to make mistakes within the limits of the systems offered by the programming language. As a result, Swift application programs are less prone to leaks and slow

© STM Journals 2024. All Rights Reserved

performance owing to hardware resource retention. It has lightweight syntax, allowing developers to spend less time writing additional logic and potentially saving up to three times the amount of code. The original Swift is faster than Python, which is widely used in developing AI systems, and features such as smart multithreading, which can manage many operations continuously, as well as a reference counting mechanism that optimizes memory consumption. It also works well with Objective C because it is a C-family language that is fully compatible and inherits almost everything from Objective C, such as function names, string syntax, namespaces, and unit-like syntax. It uses the Objective C runtime library to improve the performance of the Objective C programs. Compared with other programming languages, Swift has a comprehensive range of features. The unique aspects of Swift are specifically designed to be applied in Xcode, an integrated development environments (IDE) for application programming on Apple devices.

AI AND MACHINE LEARNING IN PROGRAMMING

The evolution of computer programming and programming languages is not possible without discussing the role of AI and machine learning in changing the traditional programming landscape. An era of programming is coming, which is going to be AI-oriented. The application of AI and machine learning to computer programming is the root cause of many new programming trends. Traditional programming paradigms have been significantly changed by the application of AI and machine learning. AI, machine learning, natural language processing, and deep learning can all be combined with programming. Machine learning and AI can be applied in various forms in programming, such as automating programming languages, developing intelligent programming languages, and decision-making and prediction-oriented programming.

AI algorithms can help deliver more intelligent code, starting from developing the code, debugging, and improving the code for the better. The development of the code would become easier than before. AI algorithms are based on historical data, which means that machine learning is a data-driven approach to solving the problems of development, debugging, and improvement of code. Several platforms and frameworks are available in the industry that use machine learning for code completion, debugging, and automated code optimization. The development language used in these platforms and frameworks is based on the limitations of available data and ethical issues in the development process.

Programmers and developers also need skills in AI and machine learning for the successful use of these platforms and frameworks. The input data used in programming practices should also be properly classified and require an understanding of AI and machine learning models, which can affect prediction and decision-making processes. Predictions can be made based on historical data. The code developed using historical data models was more efficient. The application of AI and machine learning in programming can affect the quality of software and development costs. In short, the future of programming is moving toward AI and machine learning.

FUTURE DIRECTIONS AND IMPLICATIONS

This essay begins with an exploration of programming language paradigms and examines trends and influences on the evolution of computer programming languages. Programming languages and paradigms are experiencing rapid development and shifts. We do not definitively know what future programming languages and paradigms will look like, but given the current trends in technology, we can make some educated guesses. We can expect to see a rise in domain-specific languages (DSLs) that are tailored to a specific task. We can also expect more multi-paradigm languages to provide the best features of all possible paradigms. We are also likely to continue to see numerous other minor shifts and trends given the rapid pace of technological change.

Some warnings to developers are in the order of language design and practice changes. First, we expect to see more automated features from software development. As AI and machine learning continue to develop and are integrated into an increasing number of tools, developers need to know that they will continue to shift. DevOps principles will also expand in both scope and application and will need to be

© STM Journals 2024. All Rights Reserved

better understood. There is also the potential for a shift in this paradigm because of omnipresent AI. The rise of quantum computing will also likely have significant implications for interspersed human learning phases as software development shifts to make room for quantum computing progress. We will also see a rise in intense community collaboration and innovation. This is also likely to significantly change the landscape of software development. Community contributions and open-source efforts are also likely to continue to shape which programming languages are used and which ones die. Future languages will need to find a balance between being general and usable enough for any purpose, while also being specialized enough to provide unique benefits that other languages do not.

CONCLUSION

In conclusion, the landscape of computer programming and languages is evolving rapidly, driven by technological advancements, industrial demands, and shifts in software development practices. Key trends include:

1. *Rise of high-level abstractions and DSLs:* As complexity increases, there is a trend toward programming languages that provide higher levels of abstraction, making it easier for developers to work on complex systems without managing low-level details. DSLs have gained popularity for their focus on specific industries, use cases, and streamlining development processes.
2. *Emergence of new paradigms:* Functional programming has seen a resurgence, emphasizing immutability and pure functions to handle concurrency and scalability more efficiently. Meanwhile, OOP remains widely used, but its dominance is challenged as multi-paradigm languages gain traction.
3. *Focus on developer productivity:* Modern programming languages prioritize productivity and offer better tooling, built-in testing, and debugging features. Automation in code generation and IDEs has led to more efficient workflow and reduced development cycles.
4. *Performance optimization:* There is a continued push for languages that balance ease of use with performance. Just-in-time (JIT) compilation, as observed in languages such as Python, Java, and JavaScript, allows developers to achieve optimized performance without sacrificing productivity.
5. *Growth of AI and machine learning integration:* As AI and machine learning have become integral to software systems, new languages, libraries, and tools are being developed to support these domains. Python, with its extensive libraries (e.g., TensorFlow and PyTorch), remains the most popular language for AI research, while other languages such as Julia and R offer specialized capabilities for data science.
6. *Emphasis on security and privacy:* Security has become a critical concern, especially in languages designed for system programming where memory safety and vulnerability protection are prioritized. Languages such as Rust are gaining adoption because of their strong guarantees of safety and performance.
7. *Web and mobile development trends:* JavaScript continues to dominate web development, but frameworks and tools, such as WebAssembly, enable new performance standards for web applications. For mobile devices, cross-platform frameworks such as Flutter and React Native are changing how developers approach app development.
8. *Sustainability and open-source collaboration:* Pen-source communities are essential to advancing programming languages. Collaborative efforts help languages evolve quickly by incorporating innovations that promote efficiency, maintainability, and sustainability in software engineering.

Ultimately, the future of programming languages is shaped by the need for more intuitive, versatile, and secure development environments. With the progress of technology, new programming languages and paradigms will continue to arise, enabling developers to tackle the challenges posed by increasingly intricate and interconnected systems.

REFERENCES

1. Pierce BC, editor. Advanced Topics in Types and Programming Languages. Cambridge (MA): MIT Press; 2024.

© STM Journals 2024. All Rights Reserved

2. Klabnik S, Nichols C. The Rust Programming Language. San Francisco: No Starch Press; 2023.
3. Roziere B, Lachaux MA, Chanussot L, Lample G. Unsupervised translation of programming languages. Adv Neural Inf Process Syst. 2020;33:20601–11.
4. Ross SI, Martinez F, Houde S, Muller M, Weisz JD. The programmer's assistant: Conversational interaction with a large language model for software development. In: Proc of the 28th International Conference on Intelligent User Interfaces; 2023 Mar 19-22; Sydney, Australia. p. 491–514. DOI: 10.1145/3581641.3584037.
5. Roziere B, Lachaux MA, Chanussot L, Lample G. Unsupervised translation of programming languages. [Preprint]. arXiv:2006.03511. 2020 Jun 5. DOI: 10.48550/arXiv.2006.03511.
6. Becker BA, Denny P, Finnie-Ansley J, Luxton-Reilly A, Prather J, Santos EA. Programming is hard – or at least it used to be: Educational opportunities and challenges of AI code generation. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023). Association for Computing Machinery; 2023 Mar 6–9; Toronto, Canada. Vol. 1. p. 500–6. DOI: 10.1145/3545945.3569759.
7. Trott CR, Lebrun-Grandié D, Arndt D, Ciesko J, Dang V, Ellingwood N, Gayatri R, Harvey E, Hollman DS, Ibanez D, Liber & Kokkos N. Programming model extensions for the exascale era. IEEE Trans Parallel Distrib Syst. 2021;33:805–17.
8. Vaithilingam P, Zhang T, Glassman EL. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In: CHI Conference on Human Factors in Computing Systems Extended Abstracts; 2022 Apr 29–May 5; New Orleans, LA, USA. p. 1–7. DOI: 10.1145/3491101.3519665.
9. Campbell-Kelly M, Aspray WF, Yost JR, Tinn H, Con Díaz GC. Computer: A history of the information machine. New York: Routledge; 2023. DOI: 10.4324/9781003263272.
10. Chen L, Guo Q, Jia H, Zeng Z, Wang X, Xu Y, Wu J, Wang Y, Gao Q, Wang J, Ye W. A survey on evaluating large language models in code generation tasks. [Preprint]. arXiv:2408.16498. 2024 Aug 29. DOI: 10.48550/arXiv.2408.16498.

© STM Journals 2024. All Rights Reserved