Amrita Vishwa Vidyapeetham

TIFAC-CORE in Cyber Security

20CYS312 - Principles of Programming Languages
Assignment-01: Exploring Programming Paradigms

«Siddharth Krishna R»

21st January, 2024

Paradigm 1: <Scripting>

Discuss the principles and concepts of Paradigm 1.

Language for Paradigm 1: <Name of Language 1>

Discuss the characteristics and features of the language associated with Paradigm 1.

Paradigm 2: <Name of Paradigm 2>

Discuss the principles and concepts of Paradigm 2.

Language for Paradigm 2: <Name of Language 2>

Discuss the characteristics and features of the language associated with Paradigm 2.

Analysis

Provide an analysis of the strengths, weaknesses, and notable features of both paradigms and their associated languages.

Comparison

Compare and contrast the two paradigms and languages, highlighting similarities and differences.

Challenges Faced

Discuss any challenges you encountered during the exploration of programming paradigms and how you addressed them.

Conclusion

Summarize your findings and conclude the assignment.

References

Include any references or sources you consulted for your assignment.

1 Scripting

Scripting is the process of writing scripts, which are sets of instructions that automate tasks. Scripts are commonly used in programming and computer science to perform various functions, ranging from simple tasks to complex operations.

1.1 Key principles of Scripting:

1. Automation:
Purpose: The primary goal of scripting is to automate repetitive tasks or processes, making them more efficient and less error-prone. Example: Writing a script to automate file backups, software installations, or data processing.

2. Interpreted Languages:

Execution: Scripting languages are often interpreted rather than compiled. This means that the script's code is executed directly by an interpreter, line by line. Examples: Python, JavaScript, Ruby, and Shell scripting languages (e.g., Bash).

3. Readability and Simplicity:

Code Clarity: Scripts are often written to be easily readable and understandable, emphasizing simplicity over complexity. Maintenance: Clear code makes it easier to maintain and update scripts over time.

4. Rapid Development:

Quick Prototyping: Scripting allows for rapid development and prototyping of solutions due to the dynamic nature of interpreted languages.

5. Task-Specific:

Scope: Scripts are typically designed to accomplish specific tasks or solve particular problems rather than to build entire applications.

6. Flexibility:

Adaptability: Scripts are adaptable and can be modified quickly to accommodate changes in requirements or environments.

7. Portability:

Platform Independence: Many scripting languages are designed to be platform-independent, allowing scripts to run on different operating systems without modification.

8. Integration:

Integration with Other Tools: Scripts are often used to integrate different software tools, services, or systems, facilitating interoperability.

9.Error Handling:

Graceful Degradation: Scripts may include mechanisms for handling errors gracefully, providing feedback or alternative actions when something goes wrong.

10. Scripting Language Choice:

Selection Criteria: Choosing the appropriate scripting language depends on factors such as task requirements, ease of use, available libraries, and community support.

11. Security Considerations:

Input Validation: Proper validation of user input and secure coding practices are crucial to prevent security vulnerabilities in scripts.

12. Debugging and Testing:

Debugging Tools: Scripting languages often come with tools for debugging, making it easier to identify and fix errors. Unit Testing: Testing frameworks are used to ensure the reliability of scripts, especially in larger projects.

1.2 Examples of Scripting Languages:

- **Bash:** It is a scripting language to work in the Linux interface. It is a lot easier to use bash to create scripts than other programming languages. It describes the tools to use and code in the command line and create useful reusable scripts and conserve documentation for other people to work with.
- **Node js:** It is a framework to write network applications using **JavaScript**. Corporate users of Node.js include IBM, LinkedIn, Microsoft, Netflix, PayPal, Yahoo for real-time web applications.
- **Ruby:** There are a lot of reasons to learn Ruby programming language. Ruby's flexibility has allowed developers to create innovative software. It is a scripting language which is great for web development.
- **Python:** It is easy, free and open source. It supports procedure-oriented programming and object-oriented programming. Python is an interpreted language with dynamic semantics and huge lines of code are scripted and is currently the most hyped language among developers.
- **Perl:** A scripting language with innovative features to make it different and popular. Found on all windows and Linux servers. It helps in text manipulation tasks. High traffic websites that use Perl extensively include priceline.com, IMDB.

1.3 Difference between a scripting language and a programming language?

In the scripting language vs programming language discussion, here's what you need to know once you have understood what is a scripting language:

The primary difference between a scripting language and a programming language is in their execution – programming languages use a compiler to convert the high-level programming languages into machine language, on the other hand, scripting languages use an interpreter. While a compiler compiles a code in a complete chunk, an interpreter compiles a code line by line.

By definition, programming language is essentially a formal language that combines a set of instructions that can be fed into the computer to generate a specific output.

A scripting language falls under the umbrella of programming languages and supports scripts that are programs written exclusively for a special runtime environment to automate the execution of a specific function.

1.4 What are the types of scripting languages?

There are two types of scripting languages. Server-side scripting language and client-side scripting language.

1.4.1 Server-side scripting languages

Server-side scripting is required to access or store persistent data like user profile information. Whether that involves pulling data from a file server, database or mail server, it can all be done with a server-side scripting language like PHP.

Web servers are used to execute server-side scripting. They are basically used to create dynamic pages. It can also access the file system residing at the web server. Server-side environment that runs on a scripting language is a web server.

Scripts can be written in any of a number of server-side scripting language available. It is used to retrieve and generate content for dynamic pages. It is used to require to download plugins. In this load times are generally faster than client-side scripting. When you need to store and retrieve information a database will be used to contain data. It can use huge resources of server. It reduces client-side computation overhead. Server sends pages to request of the client.

Server-side uses

- It processes the user input
- Displays the requested pages
- Structure of web applications
- Interaction with servers/storages
- Interaction with databases
- Querying the database
- Encoding of data into HTML
- Operations over databases like delete, update.

Examples of server-side scripting languages

- PHP
- ASP.NET (C# OR Visual Basic)
- C++
- Java and JSP
- Python
- Ruby on Rails and so on.

1.4.2 Client-side scripting languages

Running in the user's browser, client-side scripting generates dynamic content by processing the code received from the server. Usually this consists of JavaScript, combined with HTML and CSS.

There are a wide range of client-side frameworks available, including React.js and Angular. Some frameworks combine client-side and server-side scripting, like Vue.js and Laravel. These work well together and can make the creation of a complete application far less painful, with lots of helpful tutorials available.

Once a page is loaded, client-side scripting doesn't require any additional help from the server – all the work is done by your browser on its own. Client-side scripting can respond to user input to display different content based on specific actions, but all the actual data has to be provided by the server beforehand.

Sites that use a lot of client-side scripting can perform well and take some of the load off the server, but it can also be difficult to implement advanced functionality using client-side techniques alone.

Client-side uses

- Makes interactive web pages
- Make stuff work dynamically
- Interact with temporary storage
- Works as an interface between user and server
- Sends requests to the server
- Retrieval of data from Server
- Interact with local storage
- Provides remote access for client-server program

Examples of client-side scripting languages

- JavaScript
- VBScript
- HTML (Structure)
- CSS (Designing)
- AJAX
- jQuery etc.

1.5 Disadvantages of Scripting

1. Complex scripts take a lot of time to create and test.

2. Scripts have to be managed and secured.

3. Scripts typically lack the coding standards followed by developers.

4. Network administrators often lack the advanced skills needed to undertake scripting for data translation and API connections.

5. Scripts lack job management capabilities.

6. Scripts lack alerts and reporting.

2 Bash

Bash, short for "Bourne Again SHell," is a command processor and scripting language widely used in Unix and Unix-like operating systems, including Linux. It is the default shell for most Linux distributions and macOS. Bash is an extension of the original Bourne Shell (sh) and incorporates features from the Korn Shell (ksh) and the C Shell (csh).

2.1 Key Aspects of Bash

1. Command-Line Interface (CLI):
   - Interactive Shell: Bash provides an interactive command-line interface where users can enter commands and receive immediate feedback.
   - Scripting: It is also a powerful scripting language, allowing users to create scripts to automate tasks and execute sequences of commands.
2. Basic Syntax:
   - Variables: Variables are defined and used without explicit declaration. For example, `my_variable="Hello"` sets the value of a variable.
   - Command Substitution: Using `$(command)` allows the output of a command to replace the command itself.
   - Quoting: Single quotes (`'`) preserve the literal value of each character, while double quotes (`"`) allow variable and command substitution.
3. Control Structures:
   - Conditionals: Bash supports if statements, case statements, and test expressions for implementing conditional logic.
   - Loops: `for`, `while`, and `until` loops enable repetitive execution of commands.
4. Functions:
   - Function Definition: Functions are defined using the `function` keyword or simply with `()`.
   - Function Invocation: Functions are called by name, and parameters can be passed.
5. Input/Output:
   - Standard Streams: Bash processes input from standard input (`stdin`), and it sends output to standard output (`stdout`) and errors to standard error (`stderr`).
   - Redirection: Output can be redirected to files (`>`, `>>`), and input can be taken from files (`<`).
   - Pipelines: Commands can be connected using pipes (`|`) to pass the output of one command as the input to another.
6. Variables and Environment:
   - Environment Variables: Bash maintains environment variables that influence the behavior of the shell and user programs.
   - User Variables: Users can create and use their own variables within scripts.
7. File Manipulation:
   - File Testing: Bash allows testing of files with operators like `-e` (existence), `-f` (regular file), and others.

- File Operations: Various commands and utilities are available for file manipulation, such as `cp`, `mv`, `rm`, and `mkdir`.

8. Job Control:

   - Background and Foreground Jobs: Users can run commands in the background using `&` and bring them to the foreground or background.
   - Job Control Commands: `bg`, `fg`, and `jobs` are used to manage background and foreground jobs.

9. Shell Scripting:

   - Script Execution: Bash scripts are typically saved with a `.sh` extension and executed using the `bash` command or by making the script executable with `chmod +x` and then running it directly.
   - Shebang: The shebang (`#!`) at the beginning of a script indicates the path to the interpreter.

2.2 Example Program

```
#!/bin/bash

# Bash script to greet the user

# Prompt the user for their name
echo "Hello! What's your name?"
read user_name

# Display a personalized greeting
echo "Nice to meet you, user_name!"
```

3 Imperative Programming

Imperative programming is a programming paradigm that focuses on describing how a program operates in terms of statements that change a program's state. In imperative programming, you explicitly write a sequence of statements that define the step-by-step operations to be performed on the data. These statements can include assignments, loops, and conditional statements.

3.1 Characteristics of imperative programming

1. State and Mutability:
   - Imperative programs maintain and modify state. The state represents the current values of variables, and the program's execution involves changing these values.
   - Mutability refers to the ability to modify the values of variables.
2. Sequencing of Statements:
   - Programs are written as a sequence of statements that are executed one after the other.
   - The order of statements matters, as it determines the flow of control in the program.
3. Control Flow: Control flow structures, such as loops (for, while) and conditionals (if-else), are used to control the execution of statements based on conditions.
4. Procedural Abstraction:
   - Programs are often organized into procedures or functions that encapsulate a series of steps to perform a specific task.
   - Procedures define reusable blocks of code and help in modularizing the program.
5. Explicit Data Manipulation:
   - Imperative programming involves explicitly specifying how data is manipulated and transformed.
   - Variables are used to store and represent data, and explicit operations are performed on these variables.
6. Examples of Imperative Languages: Common programming languages that follow the imperative paradigm include C, C++, Java, Python (to some extent), and many others.

3.2 Comparisons

1. Imperative Programming vs. Declarative Programming:
   - Imperative: Focuses on describing how to achieve a result through a sequence of statements. Emphasizes explicit control flow, mutable state, and step-by-step execution.
   - Declarative: Focuses on specifying what should be achieved without necessarily specifying how to achieve it. Emphasizes expressing the logic and constraints of a problem rather than the control flow.
2. Imperative Programming vs. Functional Programming:
   - Imperative: Uses statements that change a program's state. Variables are mutable, and the emphasis is on procedures and actions.
   - Functional: Treats computation as the evaluation of mathematical functions. Avoids changing state and mutable data. Emphasizes immutability, pure functions, and higher-order functions.

3. Imperative Programming vs. Object-Oriented Programming (OOP):
    - Imperative: Organizes code into procedures or functions. Focuses on actions and procedures for manipulating data.
    - Object-Oriented: Organizes code into objects, which encapsulate data and behavior. Emphasizes the modeling of real-world entities and relationships between objects.
4. Imperative Programming vs. Logic Programming:
    - Imperative: Specifies how to perform computations through sequences of statements. Involves mutable state and explicit control flow.
    - Logic: Focuses on expressing facts and rules about relationships in the form of logic clauses. The emphasis is on declaring relationships, and the interpreter determines how to achieve the results.
5. Imperative Programming vs. Event-Driven Programming:
    - Imperative: Specifies the sequence of steps to be executed. Control flow is typically determined by the order of statements.
    - Event-Driven: Responds to events or signals, such as user interactions or system events. The flow of control is driven by events, and callbacks are often used.
6. Imperative Programming vs. Procedural Programming:
    - Imperative: Describes the sequence of steps to achieve a result. Procedures are a common organizational unit.
    - Procedural: Emphasizes procedures or routines. The focus is on grouping a set of instructions into procedures for better modularity.
7. Imperative Programming vs. Dataflow Programming:
    - Imperative: Control flow is driven by statements executed in sequence.
    - Dataflow: Execution is driven by the availability of input data. Emphasizes the flow of data through a network of interconnected processing nodes.
8. Imperative Programming vs. Constraint Programming:
    - Imperative: Specifies explicit steps for computation. Mutable state and control flow are prominent.
    - Constraint: Specifies relationships between variables using constraints. The emphasis is on expressing dependencies and relationships, and a solver determines solutions.

4 Rust

Rust is a statically-typed, systems programming language that is designed to be fast, reliable, and safe. It was developed by Mozilla and has gained popularity for its emphasis on memory safety without sacrificing performance.

4.1 Characteristics of Rust:

1. Ownership System:
   - Feature: Borrow checker and ownership model.
   - Explanation: Rust's ownership system ensures memory safety by enforcing strict rules about ownership, borrowing, and lifetimes. This eliminates common issues like null pointer dereferences and data races.
2. Zero-Cost Abstractions:
   - Feature: Focus on performance.
   - Explanation: Rust provides high-level abstractions without sacrificing performance. The compiler generates code with similar efficiency to that of low-level languages like C and C++.
3. Borrowing and References:
   - Feature: Borrowing and references instead of traditional garbage collection.
   - Explanation: Rust uses a system of borrowing and references to manage memory without resorting to garbage collection. This allows for fine-grained control over memory and avoids runtime overhead.
4. Ownership Transfer:
   - Feature: Move semantics.
   - Explanation: Rust allows ownership of data to be transferred between scopes, avoiding the need for deep copying and improving performance.
5. Concurrency without Data Races:
   - Feature: Ownership model ensures thread safety.
   - Explanation: Rust's ownership system ensures that data races are impossible, making it easier to write concurrent code without introducing common concurrency bugs.
6. Pattern Matching:
   - Feature: Powerful pattern matching syntax.
   - Explanation: Rust provides a comprehensive pattern matching syntax through the `match` keyword, making it expressive and allowing for concise and readable code.
7. Error Handling:
   - Feature: Result and Option types.
   - Explanation: Rust uses the `Result` and `Option` types for explicit and safe error handling, avoiding the use of exceptions and promoting more predictable code.
8. Trait System:
   - Feature: Trait-based generics system.

- Explanation: Rust's trait system enables code reuse and polymorphism without sacrificing performance. Traits are similar to interfaces in other languages but with additional capabilities.

9. Memory Safety without Garbage Collection:
   - Feature: No garbage collector.
   - Explanation: Rust achieves memory safety through its ownership system and borrowing, eliminating the need for garbage collection. This makes Rust suitable for systems programming where manual memory management is essential.
10. Cross-Platform Support:
    - Feature: Cross-platform compilation.
    - Explanation: Rust supports cross-compilation, allowing developers to write code on one platform and compile it for various target platforms without modification.
11. C Interoperability:
    - Feature: Interoperability with C.
    - Explanation: Rust has seamless interoperability with C, enabling integration with existing C codebases and libraries.
12. Community and Ecosystem:
    - Unique Aspect: Engaged and welcoming community.
    - Explanation: Rust has a strong and welcoming community, and its development is driven by open-source collaboration. The community values safety, performance, and helping newcomers.

4.2 Example Program

```
use std::io;

    fn main() {
// Prompt the user for input
println!("Enter a number:");

// Read the user's input
let mut input = String::new();
io::stdin().read_line(&mut input).expect("Failed to read line");

// Parse the input as a floating-point number
let number: f64 = match input.trim().parse() {
Ok(num) => num,
Err(_) => {
println!("Invalid input. Please enter a valid number.");
return;
}
};

    // Calculate the square of the number
let square = number * number;

    // Print the result
println!("The square of  is: ", number, square);
}
```

5 Analysis

5.1 Scripting with Bash

5.1.1 Strengths of Scripting with Bash:

1. Ease of Use: Bash scripts are easy to write and understand, especially for simple tasks and system automation.
2. Rapid Development: Bash allows for quick prototyping and the development of scripts due to its concise syntax.
3. Powerful Command-Line Integration: Bash scripts can leverage the full power of the command line, allowing seamless integration with existing command-line tools and utilities.
4. Extensive UNIX Tooling: Bash benefits from a rich set of UNIX utilities and commands, making it powerful for text processing, file manipulation, and system administration tasks.
5. Portability: Bash scripts are usually portable across different UNIX-like operating systems, allowing for consistent behavior on various platforms.
6. Task Automation: Bash is well-suited for automating repetitive tasks, making it a preferred choice for system administrators and DevOps professionals.
7. System Interaction: Bash provides robust facilities for interacting with the underlying system, including process management, file operations, and environment variables.
8. Scripting for System Boot and Initialization: Bash is often used for writing startup scripts and system initialization scripts on UNIX-based systems.

5.1.2 Weaknesses of Scripting with Bash:

1. Limited Data Structures: Bash has limited support for complex data structures, making it less suitable for certain algorithmic or data-centric tasks.
2. Error Handling: Error handling in Bash scripts can be challenging, and scripts may not always handle errors gracefully.
3. Portability Challenges: While Bash is widely available, there might be variations between different implementations, which can lead to portability issues.
4. Performance: Bash might not be as performant as compiled languages for certain computational tasks, especially those involving heavy data processing or complex algorithms.
5. Code Readability: As scripts grow in size, maintaining readability can become challenging, especially when dealing with nested structures or complex logic.
6. Limited Scope for Large-Scale Applications: Bash is primarily designed for scripting and automation and may not be the best choice for developing large-scale applications.

5.1.3 Notable Features of Scripting with Bash:

1. Command Substitution: Bash allows command substitution, enabling the use of command output as part of variable assignment or command-line arguments.
2. Pipeline and Redirection: The ability to pipe commands together (|) and redirect input/output (>, <) provides powerful capabilities for data manipulation.

3. Variables and Environment: Bash allows the use of variables and environment variables to store and retrieve information.

4. Conditional Constructs: Bash supports various conditional constructs, including `if`, `elif`, and `case`, enabling the execution of different code blocks based on conditions.

5. Loops: Bash provides `for` and `while` loops for iterating over lists or until a certain condition is met.

6. Functions: Bash supports the definition and use of functions, promoting modularity in script development.

7. Job Control: Bash allows for job control, managing background and foreground processes within a script.

5.2 Imperative with Rust

5.2.1 Strengths:

1. Memory Safety:

   - Strength: Rust's ownership system ensures memory safety without sacrificing performance.
   - Analysis: The borrow checker and ownership model eliminate common memory-related issues such as null pointer dereferences and data races.

2. Concurrency without Data Races:

   - Strength: Rust allows for concurrent programming without the risk of data races.
   - Analysis: The ownership system and borrowing model enforce thread safety, making it easier to write concurrent and parallel code with confidence.

3. Zero-Cost Abstractions:

   - Strength: Rust provides high-level abstractions without sacrificing performance.
   - Analysis: Developers can use high-level constructs like iterators and closures, and the compiler generates efficient machine code, making Rust suitable for systems programming.

4. C Interoperability:

   - Strength: Rust has seamless interoperability with C.
   - Analysis: This feature enables easy integration with existing C codebases and libraries, making it practical for projects that require interaction with C components.

5. Pattern Matching:

   - Strength: Rust's powerful pattern matching syntax.
   - Analysis: Pattern matching enhances code readability and conciseness, allowing developers to express complex conditional logic in a clear and structured manner.

6. Trait System:

   - Strength: Rust's trait system for code reuse and polymorphism.
   - Analysis: Traits allow developers to achieve polymorphism and code reuse without sacrificing performance. They play a crucial role in building generic and flexible abstractions.

5.2.2 Weaknesses:

1. Learning Curve:
   - Weakness: Rust has a steeper learning curve, especially for developers transitioning from languages with garbage collection.
   - Analysis: Understanding ownership, borrowing, and lifetimes can be challenging for newcomers, but this complexity is crucial for achieving Rust's safety guarantees.
2. Verbosity:
   - Weakness: Rust code can be more verbose compared to some higher-level languages.
   - Analysis: The need for explicit lifetime annotations, borrowing syntax, and ownership rules can make Rust code appear more verbose, especially for those accustomed to more concise syntax.
3. Limited Ecosystem:
   - Weakness: Rust's ecosystem, while growing, is still smaller than more established languages like C++ or Python.
   - Analysis: Developers may find fewer libraries and frameworks available for certain domains, which could impact productivity and result in more manual implementation.

5.2.3 Notable Features:

1. Ownership System:
   - Feature: Rust's ownership system, including borrowing and lifetimes.
   - Analysis: The ownership system is a unique and central feature of Rust, providing memory safety guarantees without relying on garbage collection.
2. Borrow Checker:
   - Feature: The Rust borrow checker.
   - Analysis: The borrow checker enforces ownership rules at compile-time, preventing common programming errors related to mutable references and lifetimes.
3. Cargo Package Manager:
   - Feature: Cargo, Rust's package manager.
   - Analysis: Cargo simplifies project management, dependency tracking, and building. It contributes to Rust's ecosystem by making it easy for developers to share and reuse code.
4. Enums and Pattern Matching:
   - Feature: Rust's enums and pattern matching.
   - Analysis: Enums and pattern matching allow for expressive and exhaustive handling of data variants, contributing to safer and more readable code.
5. Concurrency Support:
   - Feature: Support for concurrent programming without data races.
   - Analysis: Rust's ownership model makes concurrent programming more approachable and safer, enabling developers to build scalable and efficient concurrent systems.

6 Comparisons

Table 1: Comparison of Scripting with Bash and Imperative Programming in Rust

Aspect	Scripting with Bash	Imperative Programming in Rust
Ease of Use	Easy to write and understand, especially for simple tasks and automation.	Requires explicit memory management, which can be more complex.
Concurrency	Limited support for concurrent programming.	Strong support for safe concurrent programming through ownership system.
Memory Safety	Less emphasis on memory safety.	Strong emphasis on memory safety through ownership and borrowing.
Performance	May have limitations for computationally intensive tasks.	Emphasizes zero-cost abstractions and high performance.
Error Handling	Error handling can be challenging.	Uses the Result and Option types for explicit and safe error handling.
Data Structures	Limited support for complex data structures.	Supports a rich set of data structures and provides flexibility for complex algorithms.
Community and Ecosystem	Engaged and welcoming community.	Growing community with a focus on safety and performance.
Portability	Generally portable across UNIX-like systems.	Portable, but may have variations across platforms.
Command Line Integration	Excellent integration with command-line tools.	Good command-line capabilities but may not be as seamless as Bash.
Expressiveness	Concise syntax for quick scripting.	Strong expressiveness with pattern matching and functional features.
Tooling	Abundant UNIX tools and utilities.	Growing ecosystem with tools and libraries available through Cargo.
Scripting vs. Application Development	Primarily designed for scripting and automation.	Suitable for both scripting and large-scale application development.
Language Paradigm	Procedural scripting language.	Multi-paradigm language with procedural, functional, and object-oriented features.

7 Challenges faced

1. Each paradigm had its own concepts and practices, and transitioning between paradigms was a bit challenging.
Couldn't explore and script all the aspects of both these paradigms since they both differed with languages.

2. The introduction of code snippets that were completely new to me(RUST) made me realize I didn't scratch the surface of both these languages. At least the concepts were clear.

3. Couldn't find many resources for Imperative programming in RUST. Uses Chatgpt, Bard to produce snippets but didn't yield a variety of unique ones.

4. Many of the features that I came across and added to the report were completely new.

8 Conclusion

In summary, scripting with Bash excels in system automation, task automation, and quick prototyping due to its ease of use and integration with UNIX tools.
However, it may have limitations in terms of data structures, error handling, and performance for certain use cases. Understanding these strengths and weaknesses helps in making informed decisions when choosing Bash for scripting tasks.
Rust's focus on memory safety, zero-cost abstractions, and performance make it a unique language suitable for a wide range of applications, including system-level programming, web development, and more.
Its ownership system, borrowing model, and emphasis on concurrency set it apart from other languages and contribute to its growing popularity.

9 References

1. geeksforgeeks
2. github
3. RUST
4. TECHTARGET
5. Youtube
6. courseera
7. stackoverflow
8. chatgpt
9. Javaatpoint
10. Forta
11. Wikipedia
12. Tutorialpoint
13. Codesdope
14. ionos
15. Educative
16. Codefresh