Shell Best Practices

Shell scripting is a powerful tool for automating tasks and managing systems. Following best practices ensures your scripts are efficient, reliable, and maintainable.

by Pratham Borghare

Importance of Consistent Coding Style

Consistent coding style enhances readability and maintainability. Use meaningful variable names, indent code consistently, and follow conventions for comments and spacing.

Readability

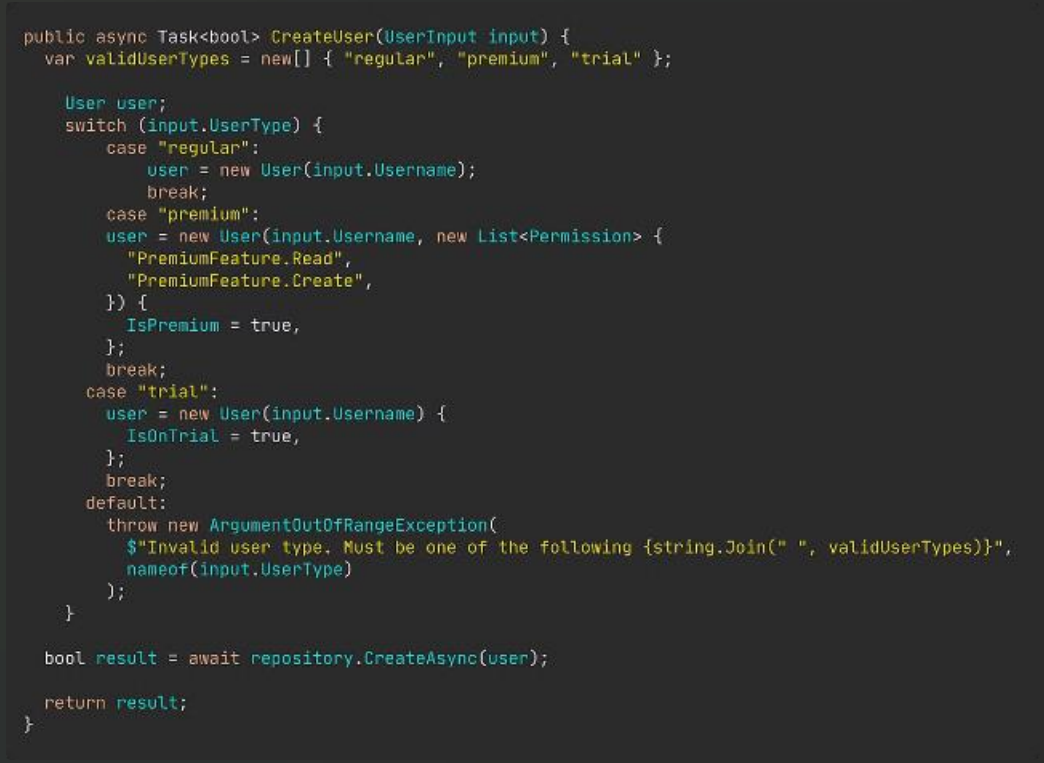
Consistent coding style makes it easier for you and others to understand your code.

Maintainability

Well-structured code is easier to modify and update.

Collaboration

Consistent style promotes collaboration by making code more accessible to other developers.



```
public async Task<bool> CreateUser(UserInput input) {
  var validUserTypes = new[] { "regular", "premium", "trial" };

    User user;
    switch (input.UserType) {
        case "regular":
            user = new User(input.Username);
            break;
        case "premium":
        user = new User(input.Username, new List<Permission> {
          "PremiumFeature.Read",
          "PremiumFeature.Create",
        }) {
          IsPremium = true,
        };
        break;
      case "trial":
        user = new User(input.Username) {
          IsOnTrial = true,
        };
        break;
      default:
        throw new ArgumentOutOfRangeException(
          $"Invalid user type. Must be one of the following {string.Join(" ", validUserTypes)}",
          nameof(input.UserType)
        );
    }

  bool result = await repository.CreateAsync(user);

  return result;
}
```

Effective Use of Variables and Functions

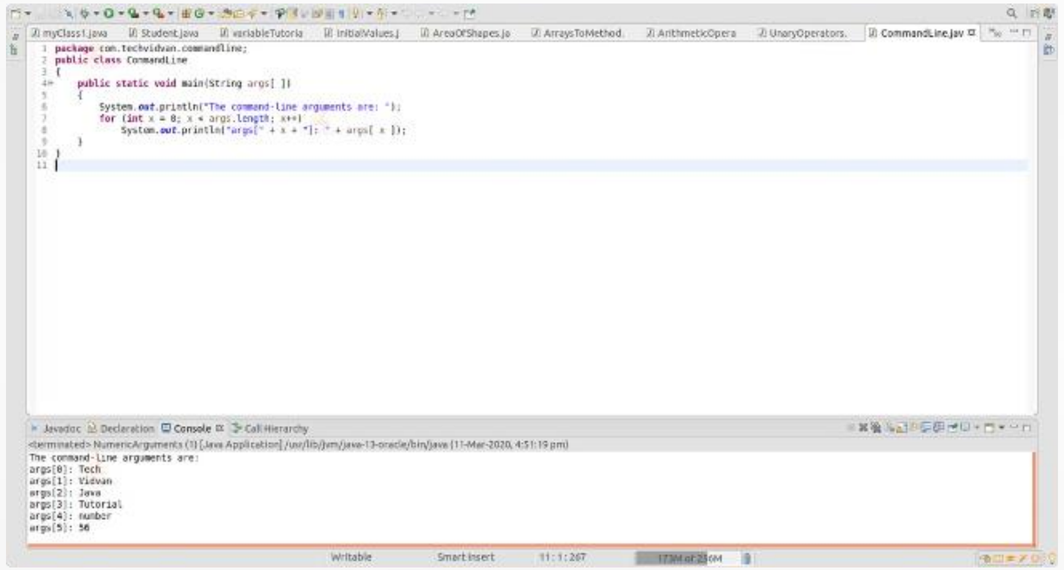
Variables store data, while functions encapsulate reusable code blocks. Use descriptive names, and avoid global variables whenever possible.

Variables

Store data values and allow you to use the same value repeatedly without having to rewrite it.

Functions

Break down complex tasks into smaller, reusable units. They make your code more modular and easier to maintain.



```
package com.techvidvan.commandline;
public class CommandLine
{
    public static void main(String args[ ])
    {
        System.out.println("The command-line arguments are: ");
        for (int x = 0; x < args.length; x++)
            System.out.println("args[" + x + "]: " + args[ x ]);
    }
}
```

```
<terminated> NumericArguments (1) [Java Application] /usr/lib/jvm/java-13-oracle/bin/java (11-Mar-2020, 4:51:19 pm)
The command-Line arguments are:
args[0]: Tech
args[1]: Vidvan
args[2]: Java
args[3]: Tutorial
args[4]: number
args[5]: 56
```

Handling Command-Line Arguments

Shell scripts can receive input through command-line arguments. Use positional arguments and named options to provide flexibility and control to users.

1

Define Arguments

Use the `$1`, `$2`, etc. variables to access positional arguments, or `$@` to access all arguments.

2

Process Arguments

Use `getopts` or similar tools to parse named options like `-h` or `--help`.

3

Use Arguments

Utilize parsed arguments to tailor your script's behavior based on user input.

Robust Error Handling and Logging

Error handling prevents unexpected script failures. Log messages to track script execution and diagnose problems.

1 **Error Detection**

Use `if`, `elif`, and `else` statements to check for potential errors.

2 **Error Messages**

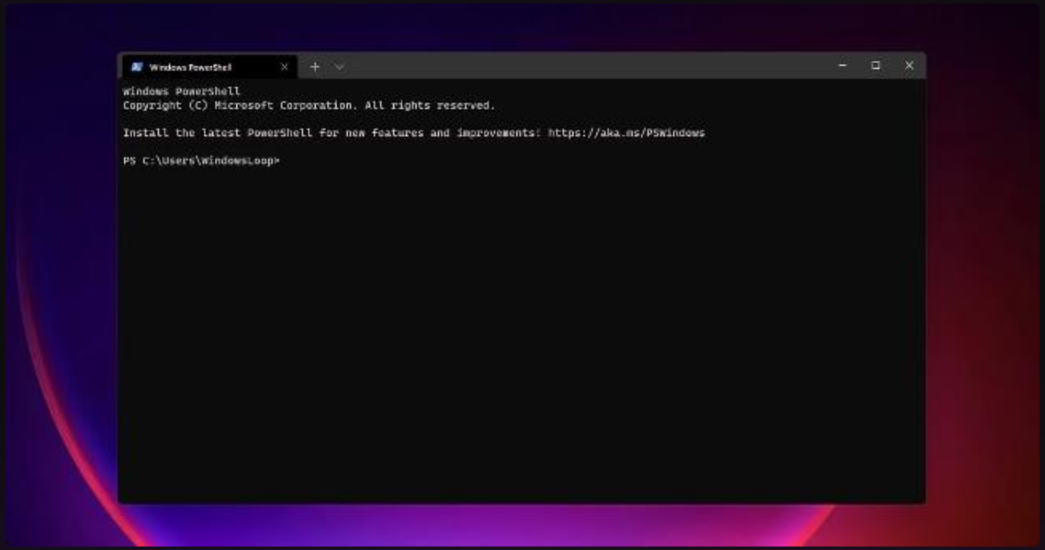
Display clear and informative error messages to help users understand the problem.

3 **Logging**

Use tools like `syslog` to record script activity for debugging purposes.

4 **Exit Codes**

Use exit codes to signal success or failure, allowing other scripts or programs to interpret the script's result.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\WindowsLoop>
```

Utilizing Conditional Statements and Loops

Conditional statements and loops allow you to control the flow of execution. Use them to make decisions based on conditions and repeat actions.

Conditional Statements	Loops
Use \`if\`, \`elif\`, and \`else\` to execute different code blocks based on conditions.	Use \`for\` and \`while\` loops to repeat actions multiple times.
Example: Check file permissions before performing an action.	Example: Iterate over a list of files and process each one.

```
git commit -a &&
cd Applications/&&
lss $HOME
```

Automating Repetitive Tasks with Shell Scripts

Shell scripts excel at automating repetitive tasks, saving you time and effort. Identify recurring tasks and write scripts to perform them automatically.

1 Identify Task

Determine the repetitive task you want to automate.

2 Write Script

Create a shell script that performs the task using commands, variables, and control structures.

3 Schedule Script

Use tools like `cron` to schedule the script to run automatically at specific intervals.

```
git commit -a &&
cd Applications/&&
lss $HOME
```

```
boot.log-20160822    httpd             pm-powersave.log   wpa_supplicant.log
boot.log-20160828    lastlog           ppp                wtmp
boot.log-20160907    maillog           prelink            Xorg.0.log
boot.log-20160911    maillog-20160822  sa                 Xorg.0.log.old
btmp                 maillog-20160828  secure             yum.log
btmp-20160901        maillog-20160907  secure-20160822    yum.log-20140318
ConsoleKit           maillog-20160911  secure-20160828    yum.log-20160613
[root@localhost ~]#
```

Optimizing Script Performance

Optimizing your scripts improves their efficiency and responsiveness. Consider using efficient commands, avoiding unnecessary processes, and optimizing for specific scenarios.

Use Efficient Commands

Select commands that perform the task efficiently, avoiding redundant steps or inefficient methods.

Minimize Memory Usage

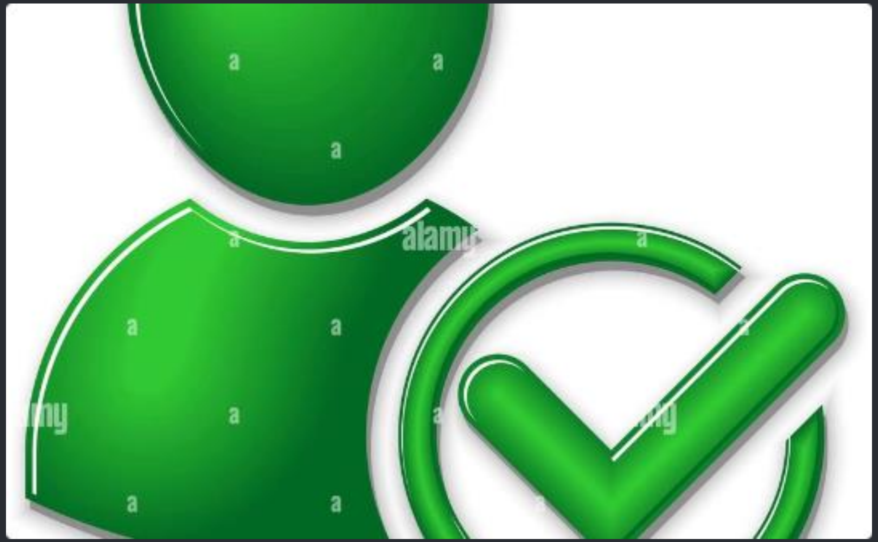
Avoid creating large temporary files or using memory-intensive commands.

Profile Script Performance

Use profiling tools to identify bottlenecks and areas for improvement.

Securing Shell Scripts and Environment

Security is paramount for shell scripts, especially those handling sensitive data. Use secure practices to protect your scripts and the systems they interact with.



Limit Permissions

Set file permissions to restrict access to your scripts, ensuring only authorized users can execute them.

Validate Input

Thoroughly validate all input received from users or external sources to prevent malicious attacks.

Use Secure Practices

Avoid using insecure commands or functions that could be exploited by attackers.

Conclusion and Additional Resources

Following these best practices enhances your shell scripting skills. Use them to write maintainable, efficient, and secure scripts.

1. **Shellcheck**

   A static analysis tool for shell scripts, identifying potential errors and vulnerabilities.

2. **Bash Guide**

   A comprehensive guide to the Bash shell, covering advanced features and best practices.

3. **Linux Command-Line Reference**

   A curated collection of Linux commands, providing descriptions and examples for common tasks.