



*"Heaven's Light is Our Guide"*

# Rajshahi University of Engineering & Technology, Rajshahi

---

## Lab Report

---

Title: Markdown Basics and Best Coding Practices

---

**Name:** Sirajum Munir

**Roll:** 2010013

**Lab Report Number:** 01

**Submission Date:** 19-10-2024

**Course Code:** ECE 3118

**Department:** Department of Electrical & Computer Engineering (ECE)

**University:** Rajshahi University of Engineering & Technology (RUET)

---

**Submitted To:**

**Oishi Joyti**

**Assistant Professor**

**Department of Electrical & Computer Engineering (ECE)**

**Rajshahi University of Engineering & Technology (RUET)**

# Markdown Cheat Sheet

---

Markdown is a lightweight markup language with plain-text formatting syntax. It's easy to learn and widely used for documentation, README files, and writing content for the web.

---

## Table of Contents

1. [Headers](#)
  2. [Text Formatting](#)
  3. [Lists](#)
  4. [Links and Images](#)
  5. [Code](#)
  6. [Blockquotes](#)
  7. [Tables](#)
  8. [Horizontal Lines](#)
  9. [Task Lists](#)
  10. [Inline HTML](#)
- 

## # Headers

Use `#` symbols to create headers. More `#` symbols create smaller headers.

```
# H1 Header
## H2 Header
### H3 Header
#### H4 Header
##### H5 Header
##### H6 Header
```

## Text Formatting

Make your text **bold**, *italic*, or ~~strikethrough~~ using these simple syntax rules:

```
**Bold Text**
*Italic Text*
***Bold and Italic Text***
~~Strikethrough Text~~
```

Example:

- **Bold Text**
- *Italic Text*
- ***Bold and Italic***
- ~~Strikethrough~~

---

## Lists

Ordered Lists (Numbered):

- ```
1. First item
2. Second item
3. Third item
```

Unordered Lists (Bullet Points):

- ```
- First item
- Second item
- Third item
```

Nested Lists:

- ```
1. First item
  - Sub-item
    - Sub-sub-item
```

---

## Links and Images

Links:

```
[Link Text](https://example.com)
```

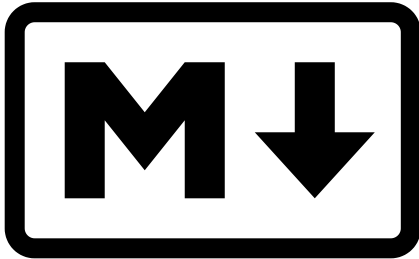
Example:

[Markdown Guide](#)

Images:

```
![Alt Text](https://via.placeholder.com/150)
```

Example:



Code

Inline Code:

```
Use `backticks` for inline code.
```

Code Blocks:

```
function helloWorld() {  
  console.log("Hello, World!");  
}
```

Example:

```
function helloWorld() {  
  console.log("Hello, World!");  
}
```

---

# Blockquotes

```
> This is a blockquote.  
> It spans multiple lines.
```

Example:

This is a blockquote.  
It spans multiple lines.

---

# Tables

Tables are useful for structuring data in rows and columns.

```
Column 1	Column 2	Column 3
Row 1	Data 1	Data 2
Row 2	Data 3	Data 4
```

Example:

| Column 1 | Column 2 | Column 3 |
|----------|----------|----------|
| Row 1    | Data 1   | Data 2   |

|       |        |        |
|-------|--------|--------|
| Row 2 | Data 3 | Data 4 |
|-------|--------|--------|

---

# Horizontal Lines

Create a horizontal line using three dashes (---), asterisks (\*\*\*), or underscores (\_\_\_).

```
---  
***  
___
```

---

# Task Lists

Use task lists to show progress or to-do items.

- [x] Completed task
- [ ] Incomplete task

Example:

- ☒ Completed task
- ☐ Incomplete task

---

## Inline HTML

Markdown allows the use of inline HTML for more advanced formatting.

```
<div style="color: red;">
  This text is red using HTML.
</div>
```

Example:

This text is red using HTML.





---

## Best Coding Practices and Conventions for Software Engineers

Good coding practices are essential for ensuring that software is maintainable, scalable, and understandable. Following conventions not only makes code cleaner but also helps others contribute efficiently.

---

### Table of Contents

- [What to Do](#) ☒
- [What Not to Do](#) 
- [How to Make Code Readable](#) 
- [Best Practices for Clear Code](#) 
- [Final Tips](#) 

---

### What to Do ☒

#### 1. Follow Consistent Naming Conventions

Stick to established naming patterns like `camelCase` for variables and `PascalCase` for classes. It

improves consistency and understanding.

```
// Good example:  
const maxLength = 50;  
class UserProfile {}
```

2. **Keep Functions and Methods Small** Break down large functions into smaller, reusable ones. Ideally, a function should perform one task only.

```
// Split large functions into smaller tasks  
func fetchData(id string) error {  
    user := getUserByID(id)  
    return sendUserData(user)  
}
```

3. **Comment Smartly** Use comments to explain why something is happening rather than what the code does. Code should explain itself when well-written.

```
# Why we're using this algorithm:  
# Optimizes the search in larger datasets
```

4. **Write Unit Tests** Always test your code. Unit tests ensure individual pieces of code work as expected and help prevent future issues.

```
func TestCalculateSum(t *testing.T) {  
    result := CalculateSum(2, 3)  
    if result != 5 {  
        t.Errorf("Expected 5 but got %d", result)  
    }  
}
```

---

## What Not to Do 🚫

1. **Avoid Hardcoding Values** Hardcoding magic numbers or strings is bad practice. Use constants or configuration files instead.

```
// Bad example:  
int maxAttempts = 5; // Magic number  
  
// Better:  
final int MAX_ATTEMPTS = 5;
```

2. **Don't Write Overly Complex Logic** If your logic is too complex, break it into smaller functions. Avoid nested loops or deeply nested `if` statements whenever possible.

```
// Bad example:
if (user.isLoggedIn && user.hasPermission && user.isAdmin) {
    // do something
}

// Better:
if (isAuthorizedUser(user)) {
    // do something
}
```

3. **Don't Skip Code Reviews** Even the best developers make mistakes. Peer reviews catch potential bugs and improve code quality.

---

## How to Make Code Readable 📖

1. **Use Descriptive Variable Names** Use meaningful variable names that describe the data they hold. Avoid abbreviations or vague terms.

```
# Good example:
user_email = "user@example.com"

# Bad example:
ue = "user@example.com"
```

2. **Consistent Indentation** Always maintain consistent indentation across your files. For example, choose 2 spaces or 4 spaces for indentation and stick to it.
3. **Group Related Code Together** Keep related code close. If functions or methods are related, place them near each other to improve readability.
4. **Use Empty Lines to Separate Logical Blocks** Use whitespace to separate different sections of your code to make it easier to scan through.

```
// Group related code
func main() {
    initializeApp()

    handleRequest()

    cleanup()
}
```



## Best Practices for Clear Code 🧠

1. **Document Your Code** Write clear and concise documentation for your public APIs, functions, and classes. This helps new developers understand the purpose and usage of the code.

```
/**
 * This function calculates the total cost of items.
 *
 * @param {Array} items - An array of items.
 * @returns {Number} - Total cost.
 */
function calculateTotal(items) {
  //...
}
```

2. **Avoid Code Duplication** DRY (Don't Repeat Yourself). Always aim to reuse code rather than duplicating it.

```
// Bad: Duplication of logic
func addUser() { ... }
func updateUser() { ... }

// Better: Reuse logic
func saveUser() { ... }
```

3. **Handle Errors Properly** Always account for edge cases and handle errors gracefully, providing useful error messages.

```
// Bad error handling:
err := saveUser()
if err != nil {
  panic(err)
}

// Better:
err := saveUser()
if err != nil {
  log.Printf("Failed to save user: %v", err)
  return
}
```

---

Here is the markdown in plain text that you can copy and paste directly:

yaml Copy code






# Best Coding Practices and Conventions for Software Engineers

---

Good coding practices are essential for ensuring that software is maintainable, scalable, and understandable. Following conventions not only makes your code cleaner but also helps others contribute efficiently.

---

## Table of Contents

- [What to Do](#) 
  - [What Not to Do](#) 
  - [How to Make Code Readable](#) 
  - [Best Practices for Clear Code](#) 
  - [Final Tips](#) 
- 

## What to Do

### 1. Follow Consistent Naming Conventions

Stick to established naming patterns like **camelCase** for variables and **PascalCase** for classes. It improves consistency and understanding.

```
// Good example:  
const maxLength = 50;  
class UserProfile {}
```

### 2. Keep Functions and Methods Small

Break down large functions into smaller, reusable ones. Ideally, a function should perform one task only.

```
// Split large functions into smaller tasks  
func fetchData(id string) error {  
    user := getUserByID(id)  
    return sendData(user)  
}
```

### 3. Comment Smartly

Use comments to explain why something is happening rather than what the code does. Code should explain itself when well-written.

```
# Why we're using this algorithm:  
# Optimizes the search in larger datasets
```

4. **Write Unit Tests** Always test your code. Unit tests ensure individual pieces of code work as expected and help prevent future issues.

```
func TestCalculateSum(t *testing.T) {  
    result := CalculateSum(2, 3)  
    if result != 5 {  
        t.Errorf("Expected 5 but got %d", result)  
    }  
}
```

---

## What Not to Do 🚫

1. **Avoid Hardcoding Values** Hardcoding magic numbers or strings is bad practice. Use constants or configuration files instead.

```
// Bad example:  
int maxAttempts = 5; // Magic number  
  
// Better:  
final int MAX_ATTEMPTS = 5;
```

2. **Don't Write Overly Complex Logic** If your logic is too complex, break it into smaller functions. Avoid nested loops or deeply nested if statements whenever possible.

```
// Bad example:  
if (user.isLoggedIn && user.hasPermission && user.isAdmin) {  
    // do something  
}  
  
// Better:  
if (isAuthorizedUser(user)) {  
    // do something  
}
```

3. **Don't Skip Code Reviews** Even the best developers make mistakes. Peer reviews catch potential bugs and improve code quality.

---

## How to Make Code Readable 👁

1. **Use Descriptive Variable Names** Use meaningful variable names that describe the data they hold. Avoid abbreviations or vague terms.

```
# Good example:
user_email = "user@example.com"

# Bad example:
ue = "user@example.com"
```

2. **Consistent Indentation** Always maintain consistent indentation across your files. For example, choose 2 spaces or 4 spaces for indentation and stick to it.
3. **Group Related Code Together** Keep related code close. If functions or methods are related, place them near each other to improve readability.
4. **Use Empty Lines to Separate Logical Blocks** Use whitespace to separate different sections of your code to make it easier to scan through.

```
// Group related code
func main() {
    initializeApp()

    handleRequest()

    cleanup()
}
```

---

## Best Practices for Clear Code 🧠

1. **Document Your Code** Write clear and concise documentation for your public APIs, functions, and classes. This helps new developers understand the purpose and usage of the code.

```
/**
 * This function calculates the total cost of items.
 *
 * @param {Array} items - An array of items.
 * @returns {Number} - Total cost.
 */
function calculateTotal(items) {
    //...
}
```

2. **Avoid Code Duplication** DRY (Don't Repeat Yourself). Always aim to reuse code rather than duplicating it.

```
// Bad: Duplication of logic
func addUser() { ... }
func updateUser() { ... }

// Better: Reuse logic
func saveUser() { ... }
```

3. **Handle Errors Properly** Always account for edge cases and handle errors gracefully, providing useful error messages.

```
// Bad error handling:
err := saveUser()
if err != nil {
    panic(err)
}

// Better:
err := saveUser()
if err != nil {
    log.Printf("Failed to save user: %v", err)
    return
}
```

---

## Final Tips

- **Use Linters and Formatters**

Automated tools like linters and formatters can help maintain consistency and catch common mistakes.

- **Write Readable Commit Messages**

Your commit messages should explain what the code changes do. Use the format: feat: added new feature for user login

- **Refactor Regularly**

Regularly review and refactor your code to improve its quality and structure.

- **Think About Performance**

Write efficient code, but don't prematurely optimize. Focus on clarity first, and improve performance when necessary.