Writing Standard Code

Understanding the importance of clean, consistent, and maintainable code

By

Dr. Ahmed Mahmoud

Introduction

- Programming is not just about writing code that works.
- It's about writing **clean**, **readable**, and **maintainable** code.
- Following coding standards ensures team **collaboration**, **scalability**, and **bug-free software**.

Introduction

- **Clean Code** that is well-structured, follows best practices, and avoids unnecessary complexity.
- **Characteristics**:

✓ Follows naming conventions (e.g., calculateTotal() instead of func1())

✓ Avoids duplication (follows the DRY principle: Don't Repeat Yourself)

✓ Has a single responsibility per function/class (follows SOLID principles)

✓ Minimizes hardcoding (uses constants or configuration files)

✓ Properly **handles errors** (no silent failures)

```
// Dirty Code
int d; // elapsed time in days
// Clean Code
int elapsedTimeInDays;
```

Introduction

- **Readable Code** that is easy to understand at a glance, even for developers who didn't write it.
- **Characteristics:**

✓ Uses meaningful variable/method names

✓ Has consistent formatting (indentation, braces, etc.)

✓ Includes comments where necessary (but avoids over-commenting)

✓ Uses whitespace and line breaks logically

✓ Follows a clear and predictable structure

```
# Unreadable Code
    x = y + z * 2
# Readable Code
    total_price = base_price + (tax_rate * base_price)
```

Introduction

- **Maintainable Code** that can be easily updated, extended, and debugged over time.
- **Characteristics**:

✓ Modular design (functions, classes, and components are decoupled)

✓ Follows design patterns (e.g., MVC, Factory, Singleton)

✓ Has automated tests (unit tests, integration tests)

✓ Uses version control properly (Git commits with clear messages)

✓ Documented APIs and dependencies

```
// Hard to Maintain
    function processData(data) {
      // 200 lines of mixed logic
    }
// Maintainable
    function validateInput(data) { ... }
    function transformData(data) { ... }
    function saveToDatabase(data) { ... }
```

What Are Coding Standards?

- Guidelines and best practices for writing programs.
- Define **naming conventions**, **formatting**, **structure**, and more.
- Make your code **professional** and **consistent**.

Rule 1 - Use Meaningful Names

Good:

```
int age = 25;
double accountBalance = 1500.75;
```

Bad:

```
int a = 25;
double x = 1500.75;
```

Names should describe the purpose of the variable or method.

Rule 2 - Follow Naming Conventions

- **Class names:** PascalCase (StudentInfo, BankAccount)
- **Method/variable names:** camelCase (calculateTotal, getUserInfo)
- **Constants:** ALL_CAPS (MAX_USERS, DEFAULT_TIMEOUT)

Rule 2 - Follow Naming Conventions

Good:

```
public class StudentInfo {
    public void printDetails() { }
    private static final int MAX_AGE = 100;
}
public class BankAccount {
    private double balance;
    public void depositAmount(double amount) { }
}
```

Use PascalCase for classes, camelCase for variables/methods, ALL_CAPS for constants.

Rule 2 - Follow Naming Conventions

Bad:

```
public class studentinfo {
    public void Print_Details() { }
    private static final int MaxAge = 100;
}
public class bank_account {
    public void DepositAmount(double Amount) { }
}
```

Use PascalCase for classes, camelCase for variables/methods, ALL_CAPS for constants.

Rule 3 - Use Proper Indentation and Braces

Good:	Bad:

`if (isValid) {`
`    System.out.println("Valid input");`
`}`
`while (count < 5) {`
`    count++;`
`}`
	`if (isValid)`
	`System.out.println("Valid input");`
	`while (count < 5)`
	`count++;`

Always use braces and indent nested code properly.

Rule 4 - Avoid Magic Numbers

Good:	**Bad:**
private static final int MAX_LOGIN_ATTEMPTS = 3; private static final double SALES_TAX = 0.07;	if (attempts > 3) { … } price = price + (price * 0.07);

Use named constants instead of hardcoded numbers.

Rule 5 - Keep Methods Short and Focused

Good:

```
public int calculateArea(int width, int height)
{
   return width * height;
}
public String formatName(String first, String last)
{
   return first + " " + last;
}
```

Bad:

```
public void doStuff(int width, int height)
 {
   int area = width * height;
   System.out.println("Area: " + area);
   writeToFile(area);
}
public void handleCustomer() {
   // multiple unrelated operations
}
```

A method should do one thing only.

Rule 6 - Use Comments Wisely

Good:	Bad:

// Check if user is active before sending email
if (user.isActive()) {
   sendEmail(user);
}
// Validate form input for mandatory fields
validateInput(form);
	// if user is active
	if (user.isActive()) { ... } // send email
	// validate input
	validateInput(form); // validate input

Comment "why", not "what".

Rule 7 - Handle Errors Gracefully

Good:	Bad:
try {	try {
int result = 10 / divisor;	int result = 10 / divisor;
} catch (ArithmeticException e) {	} catch (Exception e) {
System.out.println("Cannot divide by zero.");	// ignored
}	}
try {	try {
FileReader reader = new FileReader("data.txt");	readFile();
} catch (FileNotFoundException e) {	} catch (Throwable t) {
System.err.println("File not found.");	// generic catch
}	}

Rule 8 - Keep Code DRY (Don't Repeat Yourself)

Good:	Bad:

`public double calculateTax(double income) {`
`    return income * TAX_RATE;`
`}`
`public void printWelcomeMessage(String user) {`
`    System.out.println("Welcome, " + user);`
`}`
	`double tax1 = income1 * 0.15;`
	`double tax2 = income2 * 0.15;`
	`System.out.println("Welcome, Alice");`
	`System.out.println("Welcome, Bob");`

Advantages of Following Standards

- Improved readability
- Easier maintenance
- Fewer bugs
- Better collaboration
- Facilitates code reuse
- Simplifies testing
- Looks professional
- Future-proof and scalable

Standard Code vs. Non-Standard Code

Standard:	Non-Standard:

public class SumCalculator {
  public static int add(int a, int b) {
    return a + b;
  }
  public static void main(String[] args) {
    int result = add(10, 20);
    System.out.println("Sum: " + result);
  }
}
	
	public class sumcalc{
	public static int Add(int A,int B){return A+B;}
	public static void main(String args[]){int
	x=10,y=20;System.out.println("sum is "+Add(x,y));}}

Summary

- Follow naming, formatting, and structural conventions.
- Keep code simple, readable, and modular.
- Use comments and constants wisely.
- Write professional code that others can understand.

Questions & Discussion

- Any questions or code you'd like to review together?
- Let's discuss real-world scenarios or your own examples!

Types of Complexity in Programming

A deep dive into time, space, and code complexity

By

Dr. Ahmed Mahmoud

What is Code Complexity?

Code complexity refers to how difficult code is to understand, maintain, or execute.

Code complexity includes measures of time, space, logic paths, readability, and algorithm efficiency.

Types of Complexity

- Time Complexity
- Space Complexity
- Cyclomatic Complexity
- Cognitive Complexity
- Structural Complexity
- Algorithmic Complexity

Time Complexity (Definition)

- Time complexity describes how the execution time of an algorithm increases with input size.
- Measured using Big-O notation, it gives the upper bound (worst-case) performance.

O(1) – Constant Time

- O(1) – Constant Time: Always takes the same amount of time, regardless of input size.
- Example → Accessing array elements:

```
int getFirst(int[] arr) {
    return arr[0];
}
```

O(log n) – Logarithmic Time

- O(log n) – Logarithmic Time: Cuts the input size in half each time.
- Example ➔ Binary search:

```
int binarySearch(int[] arr, int key) {
    ...
}
```

O(n) – Linear Time

- O(n) – Linear Time: Time grows proportionally with input size.
- Example ➔ Loop through array:

```
for (int i : arr) {
    System.out.println(i);
}
```

O(n log n) – Linearithmic Time

- O(n log n) – Linearithmic Time: Logarithmic operation performed n times.
- Example ➔ Merge sort:

```
mergeSort(arr, left, right);
```

$O(n^2)$ – Quadratic Time

- $O(n^2)$ – Quadratic Time: Performance degrades rapidly with larger inputs due to nested loops.
- Example ➔ Bubble sort:

```
for (i)
    for (j) {...}
```

$O(n^3)$ – Cubic Time

- $O(n^3)$ – Cubic Time: Three nested loops; even slower than quadratic for large inputs.
- Example ➔ Triple nested loops:

```
for (i)
    for (j)
        for (k) {...}
```

$O(2^n)$ – Exponential Time

- $O(2^n)$ – Exponential Time: Time doubles with each input increment.
- Example ➔ Recursive Fibonacci:

```
int fib(int n) {
        return fib(n-1) + fib(n-2);
}
```

O(n!) – Factorial Time

- O(n!) – Factorial Time: Explores all possible combinations/permutations.
- Example ➔ Generating permutations:

```
permute(str, result);
```

Space Complexity

Space complexity measures how much memory an algorithm uses relative to input size. Includes input storage, auxiliary variables, and recursion stack.

Cyclomatic Complexity

- Measures the number of independent paths in code.
- Formula ➔ : CC = E - N + 2P
    - E = Number of edges (control flow transitions)
    - N = Number of nodes (sequential statements)
    - P = Number of connected components (usually 1 for a single function)
- Example ➔ If/Else conditions increase cyclomatic complexity.

Simplified Formula (for a single function):

CC = Number of decision points (if, for, while, case) + 1

Cognitive Complexity

- Measures how difficult code is to understand.
- Considers nesting, recursion, jumps in logic.
- More human-centered than cyclomatic complexity.

Structural Complexity

- Refers to how modular, maintainable, and clean the code is.
- Affected by code smells, duplicate code, and poor design patterns.

Algorithmic Complexity

- Overall theoretical performance (time and space).
- Combines knowledge of algorithms, data structures, and mathematical modeling.

Summary of Complexities

- O(1): Constant
- O(log n): Logarithmic
- O(n): Linear
- O(n log n): Linearithmic
- $O(n^2)$: Quadratic
- $O(n^3)$: Cubic
- $O(2^n)$: Exponential
- O(n!): Factorial

Big-O Time Complexity Definitions

- $O(1)$ – Constant Time: Always takes the same amount of time, regardless of input size.
- $O(\log n)$ – Logarithmic Time: Cuts the input size in half each time.
- $O(n)$ – Linear Time: Time grows proportionally with input size.
- $O(n \log n)$ – Linearithmic Time: Logarithmic operation performed n times.
- $O(n^2)$ – Quadratic Time: Performance degrades rapidly with larger inputs due to nested loops.
- $O(n^3)$ – Cubic Time: Three nested loops; even slower than quadratic for large inputs.
- $O(2^n)$ – Exponential Time: Time doubles with each input increment.
- $O(n!)$ – Factorial Time: Explores all possible combinations/permutations.