**Java Coding Best Practice**

**Reference links**

<https://www.codejava.net/coding/10-java-core-best-practices-every-java-programmer-should-know>

<https://www.slideshare.net/maheshm1206/coding-standards-for-java?from_action=save>

**Ordering Class Members by Scopes**

The best practice to organize member variables of a class by their scopes from most restrictive to least restrictive.

public class StudentManager {

    private String errorMessage;

    private int numberOfColumns;

    private int numberOfRows;

    float columnWidth;

    float rowHeight;

    protected String[] columnNames;

    protected List<Student> listStudents;

    public int numberOfStudents;

    public String title;

}

**Avoid** using for loops with indexes, try to use enhanced for loop because index variable can be changed.

**Using** Interface References to Collections.

**Avoid** Redundant Initialization (0-false-null)

It’s very unnecessary to initialize member variables to the following values: 0, false and null. Because these values are the default initialization values of member variables in Java

**Using** Enums or Constant Class instead of Constant Interface

1. It’s a very bad idea to create an interface which is solely for declaring some constants without any methods. Here’s such an interface:
2. It’s because the purpose of interfaces is for inheritance and polymorphism, not for static stuffs like that. So the best practice recommends us to use an enum instead.
3. In a complex project, we can have a class which is dedicated to define constants for the application

**Using** StringBuilder or StringBuffer instead of String Concatenation

String concatenation (+) creates many intermediate object while StringBuilder & StringBuffer doesn’t create intermediate object.

**Avoid** Empty Catch Blocks

It’s a very bad habit to leave catch blocks empty, as when the exception is caught by the empty catch block, the program fails in silence, which makes debugging harder.

**Using** Underscores in Numeric Literals

This little update from Java 7 helps us write lengthy numeric literals much more readable.

int maxUploadSize = 20\_971\_520;

long accountBalance = 1\_000\_000\_000\_000L;

float pi = 3.141\_592\_653\_589F;

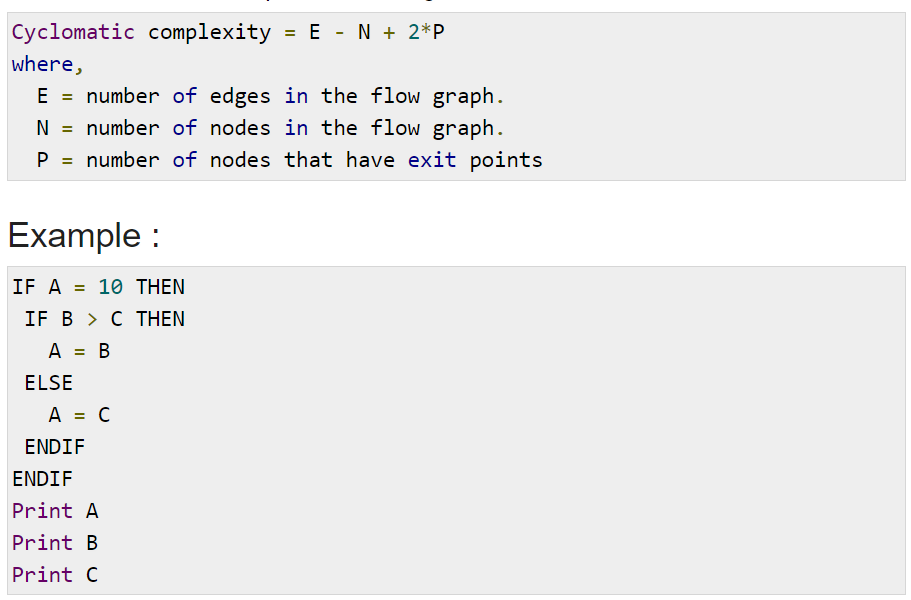
**Code Complexity**

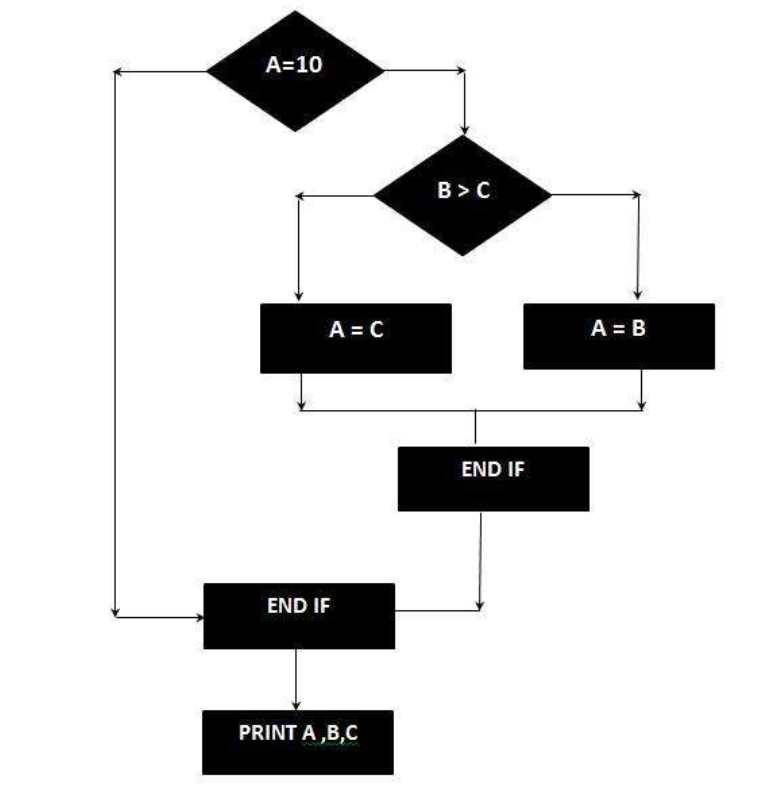
**Cyclomatic Complexity? -** Reference link:

<https://www.tutorialspoint.com/software_testing_dictionary/cyclomatic_complexity.htm>

<https://www.axelerant.com/resources/team-blog/reducing-cyclomatic-complexity-and-npath-complexity-steps-for-refactoring>

<https://www.infoworld.com/article/2074995/dealing-cyclomatic-complexity-in-java-code.html>

Cyclomatic complexity is a source code complexity measurement that is being correlated to a number of coding errors. **Formula** 



The Cyclomatic complexity is calculated using the above control flow diagram that shows seven nodes(shapes) and eight edges (lines), hence the cyclomatic complexity is 8 - 7 + 2 = 3

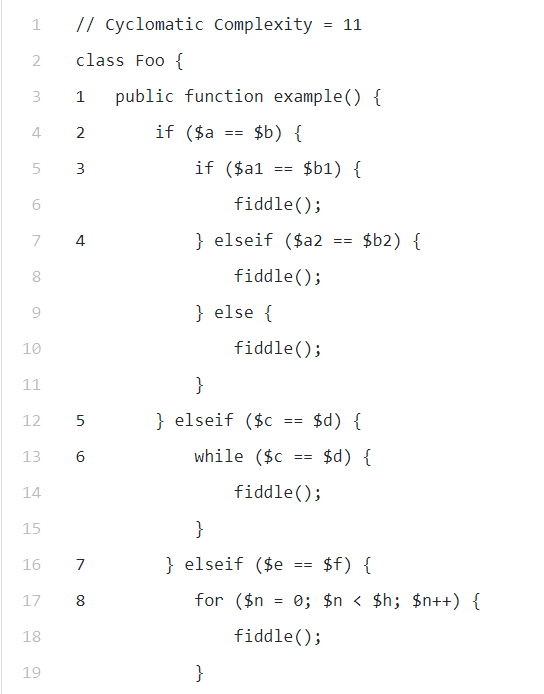
Key Points: Method should not have more than IF AND single ELSE

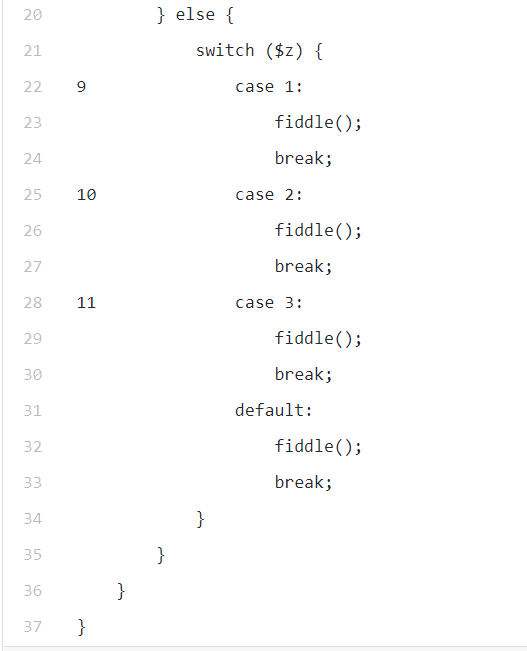
**Reducing Cyclomatic Complexity And NPath Complexity: Steps For Refactoring**

1. **Cyclomatic Complexity:**

**Class/Method:** To calculate the cyclomatic complexity of a method, each decision point increases the cyclomatic complexity of the method by one.

Notice the numbers to the left of the lines. Give one for the function declaration and add one for each decision point. The final number is the cyclomatic complexity of that method.

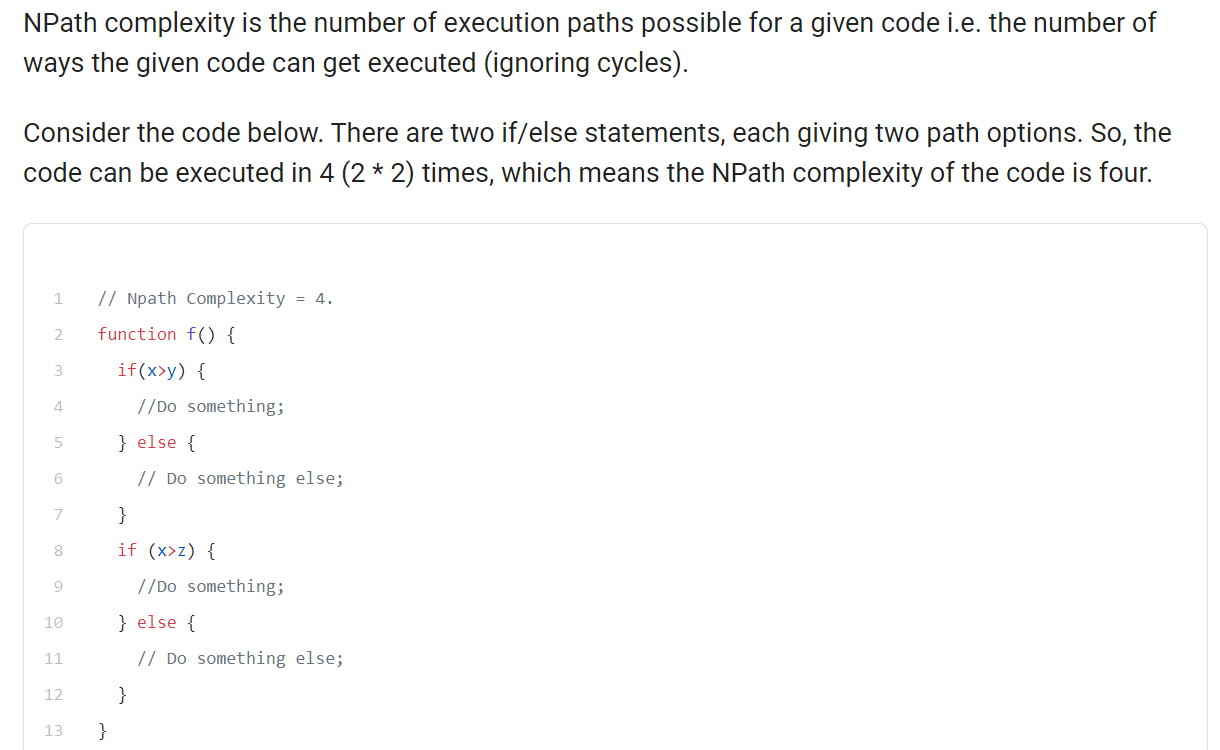




Note:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Performance/Quality | Poor | Avg | **Good** | Excellent | **Comments** |
| Complexity/Method | >7 | 4-7 | 3-4 | <3 | Maximum 3 decision points |
| Complexity/Class | >25 | 12-25 | 10-12 | <10 | Maximum 10-11 method should be within a class |
| Complexity/File | >23 | 10-23 | 8-10 | <8 |  |

1. **NPath Complexity**



**Reducing Cyclomatic Complexity**

