Source code – formatting and (self-)documentation

Marin Delchev
Marin Shalamanov
Petar Petrov
Philip Yankov

Formatting

- Why do we need it?

Why Code Needs Formatting?

```
public   const    string                    FILE_NAME
="example.bin"  ;  public static void main   (string…    a ){
FileStream   fs=     new FileStream(FILE_NAME,FileMode
.   CreateNew)   // Create the writer      for data  .
;BinaryWriter w=new BinaryWriter     (    fs      );// Write data
to                          Test.data.
for(  int i=0;i<11;i++){w.write((int)i);}w   .Close();
fs   .   close  (   ) // Create the reader    for data.
;fs=new FileStream(FILE_NAME,FileMode.          Open
,  FileAccess.Read)     ;BinaryReader              r
= new BinaryReader(fs);  // Read data from  Test.data.
 for (int i = 0; i < 11; i++){ System .out     .print
(r.read                                  ())
;}r      .    close   (   );  fs .  close  (  )  ;  }
```

Code Formatting Fundamentals

- Good formatting goals
  - To improve code readability and maintainability
- Fundamental principle of code formatting:
  - THE FORMATTING of the source code should disclose its logical structure.
  - Any formatting style that follows the above principle is good
  - Any other formatting is not good

Formatting Blocks

- Put { and } alone on a line under the corresponding parent block – in languages that have such code convention
- Indent the block contents by a single [Tab] or 4(3) spaces

```
if (some condition) {
    // Indented block contents
}
```

Empty Lines between Methods

- Use empty line for separation between methods:

```
public class FactorialCalc {
  private static long fact(int num) {
    if (num == 0)
      return 1;
    else
      return num * fact(num - 1);
  }

  public static void main() {
    long factorial = fact(5);
    System.out.println(factorial);
  }
}
```

Methods Indentation

- Methods should be indented with a single [Tab] from the class body
- Methods body should be indented with a single [Tab] as well

```
public class Indent {
   private int Zero() {
      return 0;
   }
}
```

Brackets in Methods Declaration

- Brackets in the method declaration should be formatted as follows:

```
private static long fact(int num)
```

- Don't use spaces between the brackets:

```
private static long fact ( int num )
private static long fact (int num)
```

- The same applies for `if`-conditions and `for`-loops:

Separating Parameters

- Separate method parameters by comma followed by a space
  - Don't put space before the comma

```
void registerUser(String username, String password)
registerUser(”ivan", ”petrov”);
```

  - Incorrect examples:

```
void registerUser(String username,String password)
void registerUser(String username ,String password)
void registerUser(String username , String password)
```

Empty Lines in Method Body

- Use an empty line to separate logically related sequences of lines:

```
private List<Report> prepareReports() {
  List<Report> reports = new List<Report>();

  // Create incomes reports
  Report incomesSalesReport = prepareIncomesSalesReport();
  reports.add(incomesSalesReport);
  Report incomesSupportReport = prepareIncomesSupportReport();
  reports.add(incomesSupportReport);

  // Create expenses reports
  Report expensesPayrollReport = prepareExpensesPayrollReport();
  reports.add(expensesPayrollReport);
  Report expensesMarketingReport = prepareExpensesMarketingReport();
  reports.add(expensesMarketingReport);

  return reports;
}
```

Formatting Types

- Formatting classes / structures / interfaces / enumerations
  - Indent the class body with a single [Tab]
  - Use the following order of definitions:
    - Constants, inner types, fields, constructors, get/set, methods
    - Static members, public members, protected members, default/ internal members, private members
  - The above order of definitions is not the only possible correct one

Formatting Conditional Statements and Loops

- Formatting conditional statements and loops
  - Always use `{ }` block after `if` / `for` / `while`, even when a single operator follows
  - Indent the block body after `if` / `for` / `while`
  - Always put a new line after a `if` / `for` / `while` block! (in the languages having such convention – for the rest – do what convention says)
  - Always put the `{` on the next line (in C#)
  - Always put the `{` on the same line (in JavaScript, Java)
  - Never indent with more than one [Tab]

Conditional Statements and Loops Formatting

- Example:

```
for (int i=0; i<10; i++) {
  System.out.println("i={0}", i);
}
```

- Incorrect examples:

```
for (int i=0; i<10; i++)
  System.out.println ("i={0}", i);

for (int i=0; i<10; i++) System.out.println ("i={0}", i);
//C#
for (int i=0; i<10; i++) {
  System.out.println ("i={0}", i);
}
```

Using Empty Lines

- Empty lines are used to separate logically unrelated parts of the source code

```
public static void PrintList(List<Integer> ints)  {
   System.out.println("{ ");
   for (int item : ints) {
      System.out.println (item);
      System.out.println (" ");
   }

   System.out.println("}");
}

public static void main() {
   // …
```

Misplaced Empty Lines – Example

- Empty lines are used to separate logically unrelated parts of the source code

```
public static void PrintList(List<Integer> ints)  {
   System.out.println ("{ ");
   for (int item : ints) {
      System.out.println (item);

      System.out.println (" ");

   }
   System.out.println ("}");
}
public static void main() {
   // ...
}
```

Breaking Long Lines

- Break long lines after punctuation
- Indent the second line by single [Tab]
- Do not additionally indent the third line
- Examples:

```
if (matrix[x, y] == 0 || matrix[x-1, y] == 0 ||
    matrix[x+1, y] == 0 || matrix[x, y-1] == 0 ||
    matrix[x, y+1] == 0)
{ …
Map.Entry<K, V> newEntry =
    new Map.Entry<K, V>(oldEntry.key,
oldEntry.value);
```

Incorrect Ways To Break Long Lines

```
if (matrix[x, y] == 0 || matrix[x-1, y] ==
  0 || matrix[x+1, y] == 0 || matrix[x,
  y-1] == 0 || matrix[x, y+1] == 0)
{ …
```

```
if (matrix[x, y] == 0 || matrix[x-1, y] == 0 ||
      matrix[x+1, y] == 0 || matrix[x, y-1] == 0 ||
      matrix[x, y+1] == 0)
{ …
```

```
Map.Entry<K, V> newEntry
  = new Map.Entry<K, V>(oldEntry
  .key, oldEntry.value);
```

Breaking Long Lines

- In C#/Java use single [Tab] after breaking a long line:

```
if (matrix[x, y] == 0 || matrix[x-1, y] == 0 ||
    matrix[x+1, y] == 0 || matrix[x, y-1] == 0 ||
    matrix[x, y+1] == 0)
{
    matrix[x, y] == 1;
}
```

- In JavaScript use double [Tab] in the carried long lines:

```
if (matrix[x, y] == 0 || matrix[x-1, y] == 0 ||
        matrix[x+1, y] == 0 || matrix[x, y-1] == 0 ||
        matrix[x, y+1] == 0) {
    matrix[x, y] == 1;
}
```

Alignments

- All types of alignments are considered harmful
  - Alignments are hard-to-maintain!
- Incorrect examples:

```
int        count    = 0;
DateTime   date     = DateTine.Now.Date;
Student    student  = new Student();
List<Student> students = new List<Student>();

matrix[x, y]               == 0;
matrix[x + 1, y + 1]       == 0;
matrix[2 * x + y, 2 * y + x] == 0;
matrix[x * y, x * y]       == 0;
```

Automated Tools

- Take advantage of your IDE to help formatting the code
  - Automatic alignment
  - Indentation
- Use Code analysis tools in your IDEs

Documentation

What is Project Documentation?

- Consists of documents and information
  - Both inside the source-code and outside
- External documentation
  - At a higher level compared to the code
  - Problem definition, requirements, architecture, design, project plans, test plans. etc.
- Internal documentation
  - Lower-level – explains a class, method or a piece of code

Programming Style

- Main contributor to code-level documentation
  - The program structure
  - Straight-forward, easy-to-read and easily understandable code
  - Good naming approach
  - Clear layout and formatting
  - Clear abstractions
  - Minimized complexity
  - Loose coupling and strong cohesion

Bad Comments – Example

```
public static List<Integer> FindPrimes(int start, int end) {
    // Create new list of integers
    List<int> primesList = new List<>();
    // Perform a loop from start to end
    for (int num = start; num <= end; num++) {
        // Declare boolean variable, initially true
        booleaн prime = true;
        // Perform loop from 2 to sqrt(num)
        for (int div = 2; div <= Math.sqrt(num); div++) {
            // Check if div divides num with no remainder
            if (num % div == 0) {
                // We found a divider -> the number is not prime
                prime = false;
                // Exit from the loop
                break;
// Continue with the next loop value
        }

        // Check if the number is prime
        if (prime) {
            // Add the number to the list of primes
            primesList.add(num);
        }
    }

    // Return the list of primes
    return primesList;
}
```

Self-Documenting Code – Example

```
public static List<Integer> FindPrimes(int start, int end) {
    List<int> primesList = new List<>();
    for (int num = start; num <= end; num++) {
        boolean isPrime = isPrime(num);
        if (isPrime) {
            primesList.add(num);
        }
    }

    return primesList;
}
```

Self-Documenting Code – Example

```
private static bool isPrime(int num) {
    bool isPrime = true;
    int maxDivider = Math.sqrt(num);
    for (int div = 2; div <= maxDivider; div++) {
        if (num % div == 0) {
            // We found a divider -> the number is not prime
            isPrime = false;
            break;
        }
    }
    return isPrime;
}
```

Bad Programming Style – Example

Uninformative variable names.

```
for (i = 1; i <= num; i++) {
    meetsCriteria[i] = true;
}
for (i = 2; i <= num / 2; i++) {
    j = i + i;
    while (j <= num) {
        meetsCriteria[j] = false;
        j = j + i;
    }
}
for (i = 1; i <= num; i++) {
    if (meetsCriteria[i]) {
        System.out.println(i + " meets criteria.");
    }
}
```

Good Programming Style

```
for (primeCandidate = 1; primeCandidate <= num; primeCandidate++) {
    isPrime[primeCandidate] = true;
}

for (int factor = 2; factor < (num / 2); factor++) {
    int factorableNumber = factor + factor;
    while (factorableNumber <= num) {
        isPrime[factorableNumber] = false;
        factorableNumber = factorableNumber + factor;
    }
}

for (primeCandidate = 1; primeCandidate <= num; primeCandidate++) {
    if (isPrime[primeCandidate]) {
        System.out.println(primeCandidate + " is prime.");
    }
}
```

Self-Documenting Code

- Code that relies on good programming style
  - To carry major part of the information intended for the documentation
- Self-documenting code fundamental principles
- The best documentation is the code itself
- Make the code self-explainable and self-documenting, easy to read and understand.
- Do not document bad code, rewrite it!

Self-Documenting Code Checklist

- Classes
  - Does the class's interface present a consistent abstraction?
  - Does the class's interface make obvious how you should use the class?
  - Is the class well named and does its name describe its purpose?
  - Can you treat the class as a black box?
  - Do you reuse instead of repeating code?

Self-Documenting Code Checklist

- Methods
  - Does each routine's name describe exactly what the method does?
  - Does each method perform one well-defined task with minimal dependencies?
- Type Names
  - Are type names descriptive enough to help document data declarations?
  - Are variables used only for the purpose for which they're named?
  - Do naming conventions distinguish among type names, enumerated types, named constants, local variables, class variables, and global variables?

Self-Documenting Code Checklist

- Others
  - Are data types simple so that they minimize complexity?
  - Are related statements grouped together?

Effective Comments

- Effective comments do not repeat the code
  - They explain it at a higher level and reveal non-obvious details
- The best software documentation is the source code itself – keep it clean and readable!
- Self-documenting code is self-explainable and does not need comments
  - Simple design, small well-named methods, strong cohesion and loose coupling, simple logic, good variable names, good formatting, …

Effective Comments – Mistakes

- Misleading comments

```
// write out the sums 1..n for all n from 1 to num
current = 1;
previous = 0;
sum = 1;
for (int i = 0; i < num; i++) {
  System.out.println("Sum = " + sum);
  sum = current + previous;
  previous = current;
  current = sum;
}
```

Effective Comments – Mistakes

- Comments repeating the code:

```
// set product to "base"
product = base;

// loop from 2 to "num"
for ( int i = 2; i <= num; i++ ) {
   // multiply "base" by "product"
   product = product * base;
}
System.out.println("Product = " + product);
```

Effective Comments – Mistakes

- Poor coding style:

```
// compute the square root of Num using
// the Bernulli approximation
r = num / 2;
while (abs(r - (num/r)) > TOLERANCE) {
   r = 0.5 * (r + (num/r) );
}
System.out.println("r = " + r);
```

- Do not comment bad code, rewrite it!

Key Points for Effective Comments

- Use commenting styles that don't break down or discourage modification

```
// Variable          Meaning
// --------          -------
// xPos ..... X coordinate position (in meters)
// yPos ..... Y coordinate position (in meters)
// zPos ..... Z coordinate position (in meters)
// radius ..... The radius of the sphere where the
                battle ship is located (in meters)
// distance ..... The distance from the start position
                (in meters)
```

- The above comments are unmaintainable

Key Points for Effective Comments

- Comment the code intent, not implementation details

```
// Scan char by char to find the command-word terminator ($)
done = false;
maxLen = inputString.length;
i = 0;
while (!done && (i < maxLen)) {
  if (inputString[i] == "$”) {
    done = true;
  } else {
    i++;
  }
}
```

Key Points for Effective Comments

- Focus your documentation efforts on the code

```
// Find the command-word terminator
foundTheTerminator = false;
maxCommandLength = inputString.length;
testCharPosition = 0;
while (!foundTheTerminator &&
    (testCharPosition < maxCommandLength)) {
    if (inputString[testCharPosition] == COMMAND_WORD_TERMINATOR)  {
        foundTheTerminator = true;
        terminatorPosition = testCharPosition;
    } else {
        testCharPosition = testCharPosition + 1;
    }
}
```

Key Points for Effective Comments

- Focus paragraph comments on the why rather than the how

```
// Establish a new account
if (accountType == AccountType.newAccount)  {
   …
}
```

- Use comments to prepare the reader for what is to follow
- Avoid abbreviations

Guidelines for Effective Comments

- Comment anything that gets around an error or an undocumented feature
  - E.g. `// This is a workaround for bug #3456`
- Justify violations of good programming style
- Use built-in features for commenting
- Don't comment tricky code – rewrite it
- Write documentation with the language official ways
  - XML comments in C#
  - JavaDoc in Java, etc.

Guidelines for Higher Level Documentation

- Describe the design approach to the class
- Describe limitations, usage assumptions, and so on
- Comment the class interface (public methods / properties / events / constructors)
- Don’t document implementation details in the class interface
- Describe the purpose and contents of each file
- Give the file a name related to its contents

C# XML Documentation

- In C# you can document the code by XML tags in special comments
  - Directly in the source code
- For example:

```
/// <summary>
/// This class performs an important function.
/// </summary>
public class MyClass { }
```

- The XML doc comments are not metadata
  - Not included in the compiled assembly

C# XML Documentation

- Visual Studio will use the XML documentation for autocomplete
  - Automatically, just use XML docs
- Compiling the XML documentation:
  - Compile with /doc the code to extract the XML doc into an external XML file
  - Use some tool to generate CHM / PDF / HTML / other MSDN-style documentation

JavaDoc Documentation

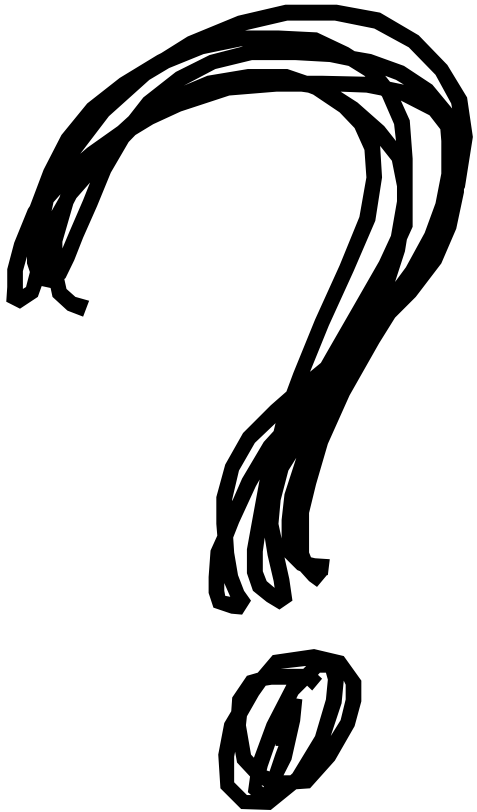
- In Java you can document the code using the built-in JavaDoc tool
  - JavaDoc tags are preceded by @ "at"
  - You can use HTML tags in the documentation code

JavaDoc Documentation

```
/**
 * This method returns the sum of two numbers
 * @param num1 The first number to sum
 * @param num2 The second number to sum
 * @return The sum of the two numbers
 */
public int sumTwoNumbers(int num1, int num2) {
}
```

JS and PHP Documentation

- JavaScript and PHP do not have official tools for code documentation. Then you have to use widely used apps for documentation (most of them are very similar to JavaDoc)



THANK YOU