# BEST CODING PRACTICES FOR OOP IN C#

by Nadia Comanici

PEAK IT - an AgileHub Event
09 december 2018

# MYTH VS REALITY

What people think programming means

Programmers
spend their time writing code

What programming actually means

Programmers
- Write code
- **Read code**
- Delete code

# CLEAN CODE

Why should we use it and why don't we use it enough?

# WHAT IS CLEAN CODE ?

„Any fool can write code
that a computer can understand.

Good programmers write code
that humans can understand”

(Martin Fowler)

# CLEAN CODE – WHY?

The way a software developer writes code
is his business card.

# RESOURCES

1. Clean Code - A Handbook of Agile Software Craftsmanship (Robert C. Martin)
2. Code Complete: A Practical Handbook of Software Construction (Steve McConnell)
3. The Pragmatic Programmer (Andrew Hunt, David Thomas)
4. https://app.pluralsight.com/library/courses/writing-clean-code-humans/
5. Start projects - https://github.com/nadiacomanici/PeakITCleanCode

# WHY IT'S IMPORTANT TO WRITE CLEAN CODE?

- Code is easier to read and understand
  - By you
  - By others
  - After months or years

- Code is easier to change
  - Less possible bugs from misunderstanding code
  - Easier maintainance

- Easier to integrate new team members

# WHY DON'T WE WRITE CLEAN CODE?

BEGINNER,
NOT ENOUGH EXPERIENCE,

HURRY,
DEADLINES,
PRESSURE

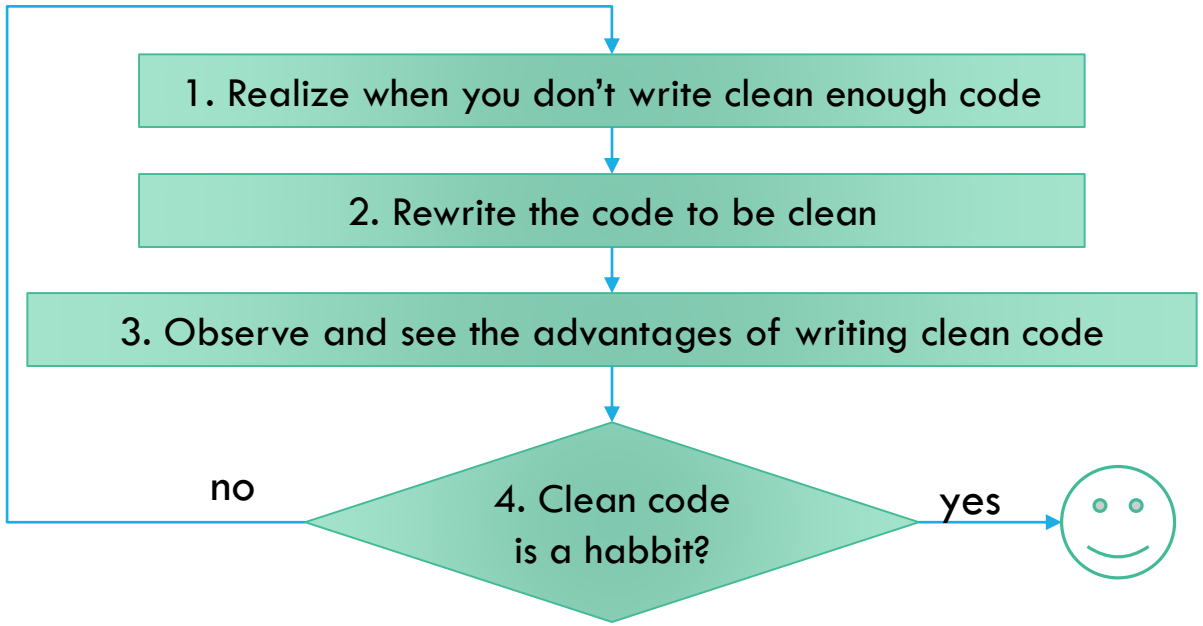
TEST CODE,
FREQUENT CHANGES
NO LONG-TERM VISION

LAZYNESS,
INDIFERENCE,
UNPROFESSIONALISM

LATER = NEVER

# HOW DO WE LEARN TO WRITE CLEAN CODE?

It's not enough to know how clean code looks like: It's like tasting food – we can say if it's good or not, but that doesn't mean we know how to prepare it to be good.



# CODE IMPROVEMENT TECHNIQUES

- ✓ Read code and compare clean code with dirty code
- ✓ Verbalize code
- ✓ Code Review
- ✓ Pair Programming
- ✓ Feedback
- ✓ Try to unit test it

# SELF DOCUMENTING CODE

- Ideally, code should be written as a book and should be easily read as a book
  - Should easily answer „What did the author want to say here?" = INTENT

- Well written code is self documented code
  - Express intent clear
  - Abstractization = shows only the important aspects and hides/ignores unnecesairy details
  - Good formatting for readability
  - Favour good code over comments

# CODING STANDARDS

# CODING STANDARDS

- A coding standard is a set of recommendations about how to write code and what are the naming and organizing conventions of code.

- Each programming language has its own coding standard, usually recommended by the authors of the language
  - Microsoft recommendations: https://msdn.microsoft.com/en-us/library/ff926074.aspx

- Coding standards can change in time and depend on the language
  - Hungarian notation is obsolete

# CODING STANDARDS FOR TEAMS

Each team should adopt a standard that best fits their needs and is appropriate to the programming language they use

Why?

- ✓ Better code quality
- ✓ Faster development and maintainance
- ✓ Easier and faster integration for new team members
- ✓ Easier to work on someone else's piece of code
- ✓ Same writing style for the entire team – proof of professionalism
  - Tools: ReSharper, FxCop, Sonar etc.

# CAMEL CASE

- C# uses Camel Case for naming variables, methods, classes etc. and Microsoft recommends the same coding standards for developers that use C#
- Camel Case means that every different word in a name should start with the first letter capitalized. This makes reading easier for the brain, because the words are easier to identify and separate inside the long sequence of letters

| Casing type | Upper Camel Case | Lower Camel Case |
|---|---|---|
| **Used for naming** | - Classes, Interfaces, Structs<br>- Methods<br>- Properties<br>- Public Fields | - Private/protected fields<br>- Local variables |
| **Examples** | `public void GetStudentById(int id)`<br>`public class Student`<br>`public string FirstName {get; private set;}` | `private List<Student> students;`<br>`string nextIndex;` |

# OOP AND SOLID PRINCIPLES

# OBJECT ORIENTED PROGRAMMING

1. Abstraction
2. Inheritance
3. Polymorphism
4. Encapsulation

- More: https://standardofnorms.wordpress.com/2012/09/02/4-pillars-of-object-oriented-programming
- Image Source: http://www.vivaxsolutions.com/web/oop.asp

# SOLID PRINCIPLES

1. Single Responsability Principle
   - Each class does a single thing
2. Open/Closed Principle
   - Each class is open for extension and closed for modification
   - An entity can allow its behavior to be extended without modifying its source code.
3. Liskov substitution Principle
   - Each object can be replaced with an object of a derived type
   - A class should not alter the contract of its parent
4. Interface Segregation Principle
   - Use smaller and multiple interfaces insted of fat interface
5. Dependency Injection Principle
   - Modules and details should depend on abstractions, not concrete classes
   - The dependencies of a class should not be created inside the class, but injected in the constructor


Image Source: http://wall-skills.com/2013/

# DEPENDENCY INJECTION - DEMO

- Scenario:
  - A class has a dependency inside it
  - The class creates an instance of the dependency, to use inside it
  - The class is reponsable for creating the dependency and managing it
- Dependency Injection (DI) usually comes hand in hand with the term „Inversion of Control" (IoC), and mean:
  - The class still has a dependency inside it
  - But the class no longer creates the instance of the dependency
  - Instead, the class receives the dependency (via constructor or a property) and just uses it as provided = DEPENDENCY INJECTION
  - So, the class no longer creates the dependency, but just asumes it will be provided and can be used, and thus the control for creating the dependency is inverted = INVERSION OF CONTROL

# DEPENDENCY INJECTION - DEMO

Evil

```
public class University
{
    // students service is a dependency for University class
    private StudentsService studentsService;

    public University()
    {
        // the university class is in control
        // because it is responsable for creating
        // the service inside its constructor
        this.studentsService = new StudentsService();
    }

    public Student GetStudentById(int id)
    {
        return studentsService.GetStudentById(id);
    }

    public Student AddStudent(string firstName, string lastName)
    {
        return studentsService.AddStudent(firstName, lastName);
    }
}
```

Good

```
public class University
{
    // StudentsService is a dependency for University class
    private StudentsService studentsService;

    public University(StudentsService studentsService)
    {
        // The university class is no longer in control
        // and no longer responsable for creating the dependency
        // Instead, it assumes it receives it and can use it
        // Inversion of Control & Dependency Injection
        this.studentsService = studentsService;
    }

    public Student GetStudentById(int id)
    {
        return studentsService.GetStudentById(id);
    }

    public Student AddStudent(string firstName, string lastName)
    {
        return studentsService.AddStudent(firstName, lastName);
    }
}
```

# ABSTRACT CLASSES VS INTERFACES

| Abstract Classes | Interfaces |
| --- | --- |
| A class that can contain constructor, fields, properties, methods, but has at least one method marked as abstract and the class is also marked as abstract | A set of methods/properties that define a contract that has to be implemented by any class that implements the interface |
| Can contain implemented members, but has to have at least one method without implemention | Just method definitions, no implementation |
| A class can inherit from a single class | A class can implement one or more interfaces |
| If a class inherits from an abstract class it can:<br>1. Implement all the abstrct members or<br>2. Remain abstract as well if it doesn't implement all methods | If a class implements an interface, it is forced to implement all the methods in that interface |
| Cannot create instances of an abstract class | Cannot create instances of an interface |

# ABSTRACT CLASSES VS INTERFACES – DEMO

```
public abstract class AbstractStudentsService
{
    protected List<Student> students;
    protected int nextId;

    public AbstractStudentsService()
    {
        this.students = new List<Student>();
        this.nextId = 1;
    }

    internal Student AddStudent(string firstName, string lastName)
    {
        Student student = new Student(nextId++, firstName, lastName);
        this.students.Add(student);
        return student;
    }

    public abstract Student GetStudentById(int id);
}

public class StudentsService : AbstractStudentsService
{
    public override Student GetStudentById(int id)
    {
        return students.SingleOrDefault(s => s.Id == id);
    }
}
```

```
public interface IStudentsService
{
    Student GetStudentById(int id);
    Student AddStudent(string firstName, string lastName);
}

public class StudentsService : IStudentsService
{
    private List<Student> students;
    private int nextId;

    public StudentsService()
    {
        this.students = new List<Student>();
        this.nextId = 1;
    }

    public Student AddStudent(string firstName, string lastName)
    {
        Student student = new Student(nextId++, firstName, lastName);
        this.students.Add(student);
        return student;
    }

    public Student GetStudentById(int id)
    {
        return students.SingleOrDefault(s => s.Id == id);
    }
}
```

# HIGH COHESION VS LOW COHESION

| High Cohesion | Low Cohesion |
| --- | --- |
| Class contains small groups of very related methods | Class contains a lot of methods that are not related to each other. |
| Methods work together with the rest of the class to accomplish the purpose of the class | Some of the methods don't use the other parts of the class |
| Easy to give names, since there is one thing that the class does | Difficult to name the class, because it does more than one thing |
| High changes to be reused because it can be easily used in other parts of the application | Most likely cannot be used in other parts of the application, so the dev might reinvent the wheel and duplicate code |
| Fields are used by multiple methods | Fields are used by a single method |

Low cohesion classes should be split into multiple smaller high cohesion classes

# COMMENTS

# COMMENTS

Comments should optimize reading and understanding code, but should not compensate for bad code.

# WHEN TO USE COMMENTS

- They have to be useful and written only when they are needed because the code itself is not enough to understand
- Up to date and reflecting what the code does
- Brief and explicative
- Clear, not misleading
- Visual Studio – Task List (todo, hack, undone)

# WHEN NOT TO USE COMMENTS

- Comments should not compensate for bad code. You should just rewrite that code
- Redundant – not DRY
- Zombie code
- Too much text that can be easily undestood from code

# WHERE TO PLACE COMMENTS?

- Always before the code block
- Indented at the same level as the code
- Space after // or /*

Good

```
// list must contain exactly 3 elements
bool isValid = (cases.size() == 3);
```

Evil

```
bool isValid = (cases.size() == 3);//list must contain 3 elements
```

# COMMENTS – ENCAPSULATION

- Easier to read and understand
- Each class should be responsable for it's own internal logic
- Encapsulation - Don't expose more then needed

Evil

```
// checks if the employee is eligible to get extra vacation days
if ((employee.Flags == WorkSchedule.FullTime) && employee.YearsOfExperience > 10)
```

Good

```
if (employee.IsEligibleForExtraVacationDays)
```

# COMMENTS – USELESS

- Funny or apology, but useless
  - Don't answer „what did the author want to say?"

Evil

```
// if you made it this far without having a nervous breakdown
// congratulations! Go get yourself a beer!

// Magic. Do not touch

// When I wrote this, only God and I understood what I was doing.
// Now, only God knows

// Bug #1234

// Sorry, this crashes and I don't know why

// Here starts the hack - See John
```

# COMMENTS – REDUNDANT

- Comments that state the same as the line of code itself
- Methods that have good names, don't need description that is actually the same as the method name (DRY)

Evil

```
// returns true
return true;

/// <summary>
/// Counts the females
/// </summary>
/// <returns></returns>
public int CountFemales()
```

# COMMENTS – MISLEADING

- Compensates bad code
- Does not reflect reality

Evil

```
public int GetRandomNumber()
{
   // chosen by rolling die
   return 4;
}
```

# COMMENTS – ZOMBIE CODE

- Unused code should not be commented – but removed
- You have the code under source control
- Makes reading more difficult and distracts attention from the active code
  - Clutters reading code with “noise code”
  - Search results might return zombie code as well
- Creates ambiguity – was the code commented by mistake?

# COMMENTS – USEFUL

Good

```
string searchString = "test";
foreach (Patient patient in this.patient)
{
   // the same string
   if (patient.Name.CompareTo(searchString) == 0)
   {
      return patient;
   }
}
```

Good

```
// Pattern explanation:
//  - "^(?:[\w]\:|\\)" -- Begin with x:\ or \\
//  - "[a-z_\-\s0-9\.]" -- valid characters are a-z| 0-9|-|.|_
//  - "(txt|gif|pdf|doc|docx|xls|xlsx)" -- Valid extension
// Matches:
//  \\192.168.0.1\folder\file.pdf
//  c:\my folder\abc abc.docx
// Non-Matches:
//  \\192.168.0.1\folder\\file.pdf
//  c:\my folder\another_folder\ab*c.v2.docx
//  file.xls
string filePattern = @"^(?:[\w]\:|\\)(\\[a-z_\-\s0-
9\.]+)+\.(txt|gif|pdf|doc|docx|xls|xlsx)$";
```

# BRACES - BLOCKS OF CODE

- The blocks of code in C# are marked with curly braces { and }
- The braces should be each on a separate line, without any other command/comment
- The braces should be specified even if there is only 1 instruction in that code block

Evil

```
public Student GetStudentById(int id) {
    return students.SingleOrDefault(s => s.Id == id);
}
```

```
if (x > 0)
    Console.WriteLine("Positive");
else Console.WriteLine("Negative or zero");
```

Good

```
public Student GetStudentById(int id)
{
    return students.SingleOrDefault(s => s.Id == id);
}
```

```
if (x > 0)
{
    Console.WriteLine("Positive");
}
else
{
    Console.WriteLine("Negative or zero");
}
```

# BRACES - CONDITIONS

- Specify the condition between round paranthesis ()

Evil

```
if x > 0
{
    Console.WriteLine("Positive");
}
```

Good

```
if (x > 0)
{
    Console.WriteLine("Positive");
}
```

# GOOD NAMES

The names you give to classes, variables, methods etc. are essential for understanding code. Even if it takes some time to „baptize” properly a variable, this will eventually save time in the future because it will be easier to understand what the variable is/does, just by taking a glance at it.

Let’s suppose you get a pet. What name would you give him?

- Dog1
- myDog
- d

Each name should represent what the variable/method/class does.

# MEANINGFUL NAMES

Give good names instead of using comments to compensate for the bad code

Evil

```
// elapsed time in days
int d;
```

Good

```
int daysSinceModification;
int fileAgeInDays;
```

# MEANINGFUL NAMES - DEMO

Give good names to understand easier the code

Evil

```
public double Compute(List<int> a)
{
    double x = 0;
    int y = 0;
    foreach (int nr in a)
    {
        x += nr;
        y++;
    }
    if (y == 0)
    {
        return 0;
    }
    else
    {
        return x / y;
    }
}
```

Good

```
public double ComputeAverage(List<int> numbers)
{
    double sum = 0;
    int howManyNumbers = 0;

    foreach (int number in numbers)
    {
        sum += number;
        howManyNumbers++;
    }

    if (howManyNumbers == 0)
    {
        return 0;
    }
    else
    {
        return sum / howManyNumbers;
    }
}
```

Good

```
public double ComputeAverage(List<int> numbers)
{
    return numbers.Count == 0 ? 0 : numbers.Average();
}
```

# BAD NAMES

- If the objects represent different things, then their names should reflect the difference
  - Don't add prefixes like: the, my, a
  - Don't add numeric suffixes

Evil

```
string theString;
string myString;
string aString;

string string1;
string string2;
string string3;
```

- Avoid too general names or prefixes

Evil

Utility
Common
...Manager
...Processor
...Info
...Data

# BAD NAMES – REDUNDANCIES

- Avoid redundant prefixes/suffixes

Evil

```
string stringName;
decimal moneyAmount;
```

Good

```
string name;
decimal money;
```

# GOOD NAMES – PREFIXES & SUFFIXES

- Avoid prefixes/suffixes that don't help you make clear differences

Evil

```
class StudentInfo
class StudentData
```

Good

```
class StudentMedicalData
class StudentSchoolData
```

Evil

```
List<Account> theList
List<Account> aList
```

Good

```
List<Account> usedAccounts
List<Account> newAccounts
```

# GOOD NAMES - SYMMETRY

- Use symmetrical naming for opposing states or prefixes

| Evil |
| --- |
| on/disable |
| quick/slow |
| lock/open |
| slow/max |

| Good |
| --- |
| on/off<br>enable/disable |
| fast/slow |
| lock/unlock<br>close/open |
| min/max |

# GOOD NAMES

- Avoid very similar names: itemInList/itemsInList
- Don't use lower „L" and „O" for naming variables
- Be consistent when giving names (use GetList all the times, not GetList sometimes and other times RetrieveList)
- Don't be a cheapskate with names – give variable names longer than a letter, because each variable has a significance
- You can use „i" and „j" for iterating through a list, but it the iterator spreads over a few tens of lines, maybe it would be better to rename the „i" and „j" so that you don't have to scroll back and see what each does.

# VERBALIZING CODE

- Avoid names that cannot be pronounced.
- Avoid abbreviations
- It will be very difficult to explain to someone else what it does

Evil

```
class DtaRcrd102
{
   private DateTime genymdhms;
   private DateTime modymdhms;
   private String pszqint = "102";
}
```

Good

```
class Customer
{
  private DateTime generationTimestamp;
  private DateTime modificationTimestamp;
  private String recordId = "102";
}
```

# GOOD NAMES IN C#

- Avoid names that start with underscore (_)
- Avoid method names that have underscore separators between the words
- Exception: the event handlers, that are generated automatically have *controlName_eventName*

Evil

```
int next_Index;
int _nextIndex;
void get_student_by_id(int id)
```

Good

```
int nextIndex;
void GetStudentById(int id)

void btnSave_Click(object sender, RoutedEventArgs e)
```

# CLASSES

# WHAT IS A CLASS?

- A class is the pattern (template) that defines a type of objects and an instance (object) is one element created using that pattern (template)
- A class contains:
  - State = fields/properties = nouns
  - Behaviour = methods = verbs

# CLASSES VS OBJECTS

A class

- The gingerbread cutter is the pattern for a gingerbread man.
- It defines the common elements for all gingerbread men:
  - 1 head, 1 mouth, 2 eyes
  - 2 hands, 2 feet, 2 buttons
- But you can't eat it

Objects (instances) of a class

- Each gingerbread man is created using that cutter and all gingerbread men have the same elements defined by the pattern:
  - 1 head, 1 mouth, 2 eyes
  - 2 hands, 2 feet, 2 buttons
- They can have some different properties
  - Coloring
  - Taste

# CLASS NAMES

- NOUNS at Singular
- As specific as possible
- Upper Camel Case
- Same for structs

Evil

```
class Students
class student
class ReadStudents
```

Good

```
class Student
class StudentsFileReader
```

# EXTRACTING CLASSES FROM TEXT - EXAMPLE

- A university has two types of persons: students and teachers.
- Each person has a first name, a last name and a birthdate.
- Each student has an identifier and some marks
- The application should display for each student the average of the marks, it he has a scholarship, if he is legally an adult and if he can vote.
- The application should allow to sort the students by last name or average mark
- In a similar manner, each teacher has a scientific title and can publish research papers

# IDENTIFYING **STATE** AND BEHAVIOUR

- A **university** has two types of **persons: students** and **teachers.**
- Each **person** has a **first name,** a **last name** and a **birthdate.**
- Each **student** has an **identifier** and some **marks**
- The **application** should display for each **student** the **average** of the **marks,** it he has a scholarship, if he is legally an adult and if he can vote.
- The **application** should allow to sort the **students** by **last name** or **average mark**
- In a similar manner, each **teacher** has a **scientific title** and can publish **research papers**

# CLASSES AFTER ANALYSIS

University

**Students**

**Teachers**

*AddStudent()

*AddTeacher()

Person

- **FirstName**
- **LastName**
- **BirthDate**
- ***Age**

Application

- DisplayStudents()
- SortStudentsByLastName()
- SortStudentsByAverageMark()

Student is a Person

- **Id**
- **Marks**
- **AverageMark**
- *HasScolarship*
- *IsLegallyAdult*
- *CanVote*
- ***FirstName**
- ***LastName**
- ***BirthDate**

Teacher is a Person

- **ScientificTitle**
- **ResearchPapers**
- ***FirstName**
- ***LastName**
- ***BirthDate**

# ENTITIES AND FILES

- Each class/struct/interface/enum should be in a separate file
- The file should have the same name as the class/struct/interface/enum
- The namespace should match the containing folder and should be updated if you move a file from a folder to another

# INTERFACE NAMES

- NOUNS at Singular, prefixed with „I”
- As specific as possible
- Upper Camel Case

Evil

```
interface Students
interface iReadStudents
interface iStudent
```

Good

```
interface IStudent
interface IStudentsFileReader
```

# METHODS

# METHOD NAMES

- Have to contain VERBS
- Each method should do one thing and one thing only
- Method names should reflect what the method does
- Method names should be descriptive, so that whoever looks at the name, understands what the method does without having to look at the implementation
- Ask yourself (or a colleague or a rubber duck)
  - What does this method do?
  - Does it do a single thing?
    - Watch out for words like: AND, OR, IF
    - If it does more, then you should split it
  - Is it in the right place?

# METHOD – SHOULD I SPLIT IT?

1. Does it fit on your screen?
   - Small methods that can be reused
   - Ideally not over 20 lines. Never over 100 lines
   - Simpler functions can be longer. Complex functions should be shorter
2. Do you use comments or white spaces to separate code inside a method?
   - You should split the code into smaller methods
3. Is this method or part of it similar to another method?
   - Avoid Duplication - Don't reinvent the wheel, just reuse it
4. Does it have a lot of indentation?
   - Cyclomatic complexity = there are many distinct paths through a method
   - Difficult to read and test, more possible bugs

# METHOD PARAMETERS

- Minimize parameters
  - 0-2 parameters – Rule of 7
  - Group them in a struct/class

Evil

```
private void SendEmail(string username, string password, string email, string message)
{
    // send email
}
```

Good

```
private void SendEmail(User user, string message)
{
    // send email
}
```

# METHODS – DO THEY DO TOO MUCH?

- Avoid flag arguments
  - Method does more than one thing

Evil

```
private void SaveUser(User user, bool sendEmail)
{
    // save user

    if (sendEmail)
    {
        // send email
    }
}
```

Good

```
private void SaveUser(User user)
{
    // save user
}

private void SendEmail(User user)
{
    // send email
}
```

# STRATEGIES FOR METHODS - DEMO

- Return early
  - If you know that certain conditions are not valid, return as fast as possible

- Fail fast – Guard clauses
  - Guard clauses – conditions at the begining of the method that assure you have all needed data before continuing
  - Guards reduce complexity and indentation
  - Guards state from the begining of the code the conditions that have to be met for the data to be valid

- Variables should have minimum lifespan
  - Declare variables only when they are needed, not from the begining of the method

# STRATEGIES FOR METHODS

Evil

```
private void btnAddNumbers_Click(object sender, RoutedEventArgs e)
{
    try
    {
        int firstNumber;
        int secondNumber;
        int sum;

        if (string.IsNullOrEmpty(txtFirstNumber.Text) == false)
        {
            firstNumber = int.Parse(txtFirstNumber.Text);
            if (string.IsNullOrEmpty(txtSecondNumber.Text) == false)
            {
                secondNumber = int.Parse(txtSecondNumber.Text);
                sum = firstNumber + secondNumber;
                textBlockSum.Text = sum.ToString();
            }
            else
            {
                throw new Exception("You must specify a value for the input numbers");
            }
        }
        else
        {
            throw new Exception("You must specify a value for the input numbers");
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message, "An error has occured", MessageBoxButton.OK, MessageBoxImage.Error);
    }
}
```

Good

```
private bool AreInputFieldsMissing()
{
    return (string.IsNullOrEmpty(txtFirstNumber.Text) == false
        || string.IsNullOrEmpty(txtSecondNumber.Text) == false);
}

private int ComputeSum()
{
    int firstNumber = int.Parse(txtFirstNumber.Text);
    int secondNumber = int.Parse(txtSecondNumber.Text);
    return firstNumber + secondNumber;
}

private void btnAddNumbers_Click(object sender, RoutedEventArgs e)
{
    try
    {
        if (AreInputFieldsMissing())
        {
            throw new Exception("You must specify a value for the input numbers");
        }

        var sum = ComputeSum();
        textBlockSum.Text = sum.ToString();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message, "An error has occured", MessageBoxButton.OK,
MessageBoxImage.Error);
    }
}
```

# CODE DUPLICATION - DEMO

- DRY Principle = Don't repeat yourself
- Avoid code duplication
  - Whenever you are tempted to copy-paste code, think if it would not be better to extract a method and use it in multiple places
  - See the common pattern for similar code and extract a method with different parameters for the differences
- Disadvantages of duplication:
  - More code to maintain
  - If you forget to change in all places -> Inconsistencies -> Bugs

# CODE DUPLICATION

Evil

```
public int CountFemales()
{
    int count = 0;
    foreach (var person in this.persons)
    {
        if (person.Gender == Gender.Female)
        {
            count++;
        }
    }
    return count;
}

public int CountMales()
{
    int count = 0;
    foreach (var person in this.persons)
    {
        if (person.Gender == Gender.Male)
        {
            count++;
        }
    }
    return count;
}
```

Good

```
private int CountByGender(Gender gender)
{
    int count = 0;
    foreach (var person in this.persons)
    {
        if (person.Gender == gender)
        {
            count++;
        }
    }
    return count;
}

public int CountFemales()
{
    return CountByGender(Gender.Female);
}

public int CountMales()
{
    return CountByGender(Gender.Male);
}
```

Better

```
private int CountByGender(Gender gender)
{
    return persons.Count(p => p.Gender == gender);
}

public int CountFemales()
{
    return CountByGender(Gender.Female);
}

public int CountMales()
{
    return CountByGender(Gender.Male);
}
```

# FIELD NAMES

- Fields represent any type of variables inside a class
- You should make the fields private or protected and make a getter/setter for it if it needs to be visible from outside the class
- According to the access modifier, the casing should be:
  - public/internal → UpperCamelCase
  - private/protected → lowerCamelCase

Evil

```
public class StudentList
{
    private List<Student> l;
    private int _next_Id;
}
```

Good

```
public class RegistrationBook
{
   private int nextId;
   private List<Person> persons;
}

public class Person
{
   private int id;
}
```

# STATIC FIELDS

- The static fields belong to the class, not to the instances
  - Static fields are shared between all instances of a class
- You don't need an instance to access a static field
- If you want to use them from outside the class, you should prefix the static field name with the class name

Good

```
public class Person
{
    public static int MinimumAgeForVoting = 18;

    //....
}

Console.WriteLine(Person.MinimumAgeForVoting);
```

# CONSTANT FIELDS

- A constant is a variable that has a known value at compile time and doesn't change its value during runtime.
- The constant fields belong to the class, not to the instances
  - constant fields are shared between all instances of a class
  - there is no need to use the "static" keyword when defining them
- Constants are used similar to static fields, by prefixing with the class name

Good

```
public class Person
{
    public const int MinimumAgeForId = 14;

    //....
}

Console.WriteLine(Person.MinimumAgeForId);
```

# READONLY FIELDS

- A readonly field is a variable that doesn't have a known value at compile time. The value is set at runtime and doesn't change its value during the lifetime of the application.
  - You should assign it when you define it or in the constructor
- The readonly fields belong to the instances, not to the class
  - You need a instance to access it and can have different values for each instance

Good

```
public class Person
{
    public readonly int MinimumAgeForRetirement;
    // ...
    private Gender gender;

    public Person(string firstName, string lastName, Gender gender)
    {
        this.MinimumAgeForRetirement = (gender == Gender.Male) ? 65 : 63;
    }
}
```

# PROPERTY NAMES

- A property is a member that allows:
  - accessing/modifying a field
  - or computing a value
- A property must have at least a get or a set, each with different access modifiers
- Properties are called similar to fields (without parenthesis, like methods)

Good

```
public class Person
{
    public int Id
    {
        get
        {
            return this.id;
        }
    }

    public string FirstName { get; private set; }
    public string LastName { get; private set; }

    public string FullName
    {
        get
        {
            return $"{FirstName} {LastName}";
        }
    }
}
```

# PROPERTY/FIELDS VISIBILITY

- When creating a field, make it private.
- When you create a property, make
  - The setter private
  - The getter internal
- Change them to something more visible only when needed

Evil

```
public class Student
{
    public string FirstName {get; set;}
}
```

Good

```
public class Student
{
    public string FirstName {get; private set;}
}
```

# BOOLEAN METHOD/PROPERTY NAMES

- Should sound like it is asking a true/false question
- Use “Is/Are/Can/Has” for a property that returns a boolean value

Evil

```
bool valid
bool start
bool open
public bool Voted
public bool GetIdCardValid()
public bool Adult
```

Good

```
bool isValid
bool hasStarted
bool isOpen
public bool CanVote
public bool HasIdCard()
public bool IsLegallyAdult
```

# COMPARE BOOLEAN

- Assign and compare implicitly

Evil

```
if (age > retirementAge)
{
    isRetired = true;
}
else
{
    isRetired = false;
}
```

Good

```
bool isRetired = (age > retirementAge);
```

- Fewer lines
- More human readable
- Less duplication
- No need to initialize before condition

Evil

```
if (isRetired == true)
```

Good

```
if (isRetired)
```

# USE POSITIVE CONDITIONALS

- It is difficult to understand double negation

Evil

```
if (!isNotValid)
```

Good

```
if (isValid)
```

# TERNARY OPERATOR

- Avoid using multiple ternaries in the same condition, it will be difficult to understand

Evil

```
if (gender == Gender.Female)
{
    retirementAge = 61;
}
else
{
    retirementAge = 65;
}
```

Good

```
int retirementAge = (gender == Gender.Female) ? 61 : 65;
```

- Fewer lines
- More human readable
- Less duplication
- No need to initialize before condition

# USE ENUMS INSTEAD OF STRINGS/MAGIC NUMBERS - DEMO

Evil

```
public void LogUser(User user)
{
    if (user.Role == "manager")
    {
        GiveAdminRights();
    }
    else
    {
        if (user.Role == "employee")
        {
            GiveEditRights();
        }
        else
        {
            GiveReadOnlyRights();
        }
    }
}
```

Good

```
public enum UserRoles
{
    Manager,
    Employee
}

public void LogUser(User user)
{
    if (user.Role == UserRoles.Manager)
    {
        GiveAdminRights();
    }
    else
    {
        if (user.Role == UserRoles.Employee)
        {
            GiveEditRights();
        }
        else
        {
            GiveReadOnlyRights();
        }
    }
}
```

Good

```
public enum Roles
{
    Manager,
    Employee
}

public void LogUser(User user)
{
    switch (user.Role)
    {
        case UserRoles.Manager:
            GiveAdminRights();
            break;
        case UserRoles.Employee:
            GiveEditRights();
            break;
        default:
            GiveReadOnlyRights();
            break;
    }
}
```

# COMPLEX CONDITIONS & ENUMS - DEMO

- Use enums instead of magic numbers
- Simplify the understanding of complex conditions by

1. Intermediate variables
2. Encapsulate in function

Evil

```
public void PlayVideo(string fileName)
{
    string fileExtension = Path.GetExtension(fileName);

    if ((fileExtension == ".mp4" || fileExtension == ".avi")
        && (CurrentState != 2 || CurrentState == 3))
    {
        LoadVideo(fileName);
        Play();
    }
}
```

Good

```
private bool IsExtensionValid(string fileName)
{
    string fileExtension = Path.GetExtension(fileName);
    return (fileExtension == ".mp4" || fileExtension == ".avi");
}

private bool CanPlayNewVideo()
{
    return (CurrentState != PlayerStates.Playing || CurrentState == PlayerStates.Paused);
}

public void PlayVideo(string fileName)
{
    if (IsExtensionValid(fileName) && CanPlayNewVideo())
    {
        LoadVideo(fileName);
        Play();
    }
}
```

# POLYMORFISM OVER ENUMS

Evil

```
public enum AttendanceTypes
{
    Daily,
    Weekend,
    Evening
}

public class Student
{
    public AttendanceTypes AttendanceType { get; private set; }

    public int GetMinimumCreditPointsToPromote()
    {
        int minCreditPoints = 0;
        switch (this.AttendanceType)
        {
            case AttendanceTypes.Daily:
                // compute and return credits for daily courses
                minCreditPoints = 100;
                break;

            case AttendanceTypes.Evening:
                // compute and return credits for evening courses
                minCreditPoints = 50;
                break;

            case AttendanceTypes.Weekend:
                // compute and return credits for weekend courses
                minCreditPoints = 35;
                break;
        }
        return minCreditPoints;
    }
}
```

Good

```
public class DailyStudent : Student
{
    public override int GetMinimumCreditPointsToPromote()
    {
        // compute and return credits for daily courses
        return 100;
    }
}

public class EveningStudent : Student
{
    public override int GetMinimumCreditPointsToPromote()
    {
        // compute and return credits for evening courses
        return 50;
    }
}

public class WeekendStudent : Student
{
    public override int GetMinimumCreditPointsToPromote()
    {
        // compute and return credits for weekend courses
        return 35;
    }
}

public abstract class Student
{
     public abstract int GetMinimumCreditPointsToPromote();
}
```

# THE POWER OF ENUMS

- Strongly type check, typos can easily generate bugs
- You can use intellisense
- You have a finite and known list of all possible values
- If you use strings and change in one place, but not all places, you will get inconsistencies (bug)
- Switch should always have a default clause

# CONSTANTS INSTEAD OF MAGIC NUMBERS

- Magic numbers are hardcoded values whose value are not obvious for someone looking the first time at the code
- They can generate bugs because:
  - Someone that doesn't know what the value represents, can change it incorrectly
  - If it is used in multiple places and the dev forgets to replace it in all the places, there will be bugs

Evil

```
return (DateTime.Now - DateOfBirth).TotalDays / 365.2425;
```

Good

```
double daysInAYear = 365.2425
return (DateTime.Now - DateOfBirth).TotalDays / daysInAYear;
```

# CONSTANTS INSTEAD OF MAGIC NUMBERS

- If it is used in multiple places and the developer forgets to replace it in all the places, there will be bugs

Evil

```
public class Person
{
    public bool CanVote
    {
        get
        {
            return Age > 18;
        }
    }

    public bool IsLegallyAdult
    {
        get
        {
            return Age > 21; //18;
        }
    }
}
```

Good

```
public class Person
{
    public static int MinimumAgeForVoting = 18;

    public bool CanVote
    {
        get
        {
            return Age > MinimumAgeForVoting;
        }
    }

    public bool IsLegallyAdult
    {
        get
        {
            return Age > MinimumAgeForVoting;
        }
    }
}
```

# REMOVE HARDCODED VALUES

- Instead of hardcoding values in the code, you might use values from the database or a configuration file
- This way, if something changes, you don't have to rebuild and install the application, just update the database – dynamic logic

Evil

```
if (yearsOfExperience > 20)
{
    return 28;
}
else if (yearsOfExperience > 10)
{
    return 23;
}
else
{
    return 21;
}
```

| Id | MinYearsOfExperience | DaysOffPerYear |
|---|---|---|
| 1 | 0 | 21 |
| 2 | 10 | 23 |
| 3 | 20 | 28 |

Good

```
GetDaysOffPerYear()
```

# EXCEPTII

# EXCEPTIONS BEST PRACTICES

- Exceptions have to be treated. Don't swallow / ignore exceptions
- If you cannot treat an exception at a level, let it go up in the call stack to a higher level for treatment
- Types of exceptions:
  1. Unrecoverable
     - The application cannot continue correctly
     - Null reference, file not found, access denied
  2. Recoverable
     - Can have temporary issues and if you try again, succeed. Have a limit for retrials
     - Retry connection, try different file, wait and try again
  3. Ignorable
     - Logging click

# QUESTIONS

Contact: nadi_comanici@yahoo.com

# FEEDBACK

https://goo.gl/forms/mzKBBK7IAE96w8P53