Object-oriented programming #1
Introduction

Wojciech Complak

Institute of Computing Science
Faculty of Computing
Poznan University of Technology

e-mail: Wojciech.Complak@wsb.poznan.pl

0.91

1

Lecture content

- the purpose and the scope of this course
- readings, online sources
- programming paradigms
- pillars of object-oriented programming paradigm
- classes: fields, methods, properties
- creating objects
- *this* reference

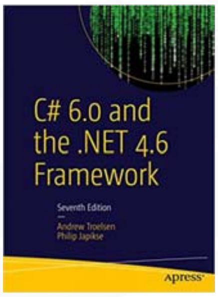
2

Purpose and scope

- presentation of the principles of object-oriented programming, based on the C# programming language,
- discussion of the principles of modelling the logical structure of IT systems using the UML language,
- introduction of the rules for constructing large-scale application software,
- comparison of principles implementation in C# with different approaches used in other popular object-oriented languages such as C++ and Java

3

Readings



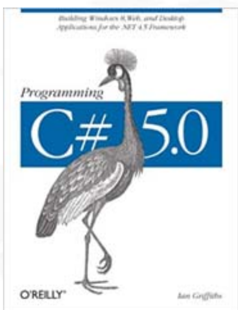
C# 6.0 and the .NET 4.6 Framework
7th Edition

Andrew Troelsen, Japikse Philip

Apress, 2015

4

Supplementary reading



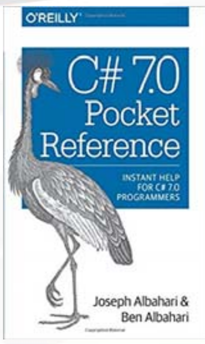
Programming C# 5.0: Building Windows 8, Web, and Desktop Applications for the .NET 4.5 Framework

Ian Griffiths

O'Reilly, 2012

5

Essential reading



C# 7.0 Pocket Reference: Instant Help for C# 7.0 Programmers

Joseph Albahari, Ben Albahari

O'Reilly, 2017

6

Object Oriented Methodology

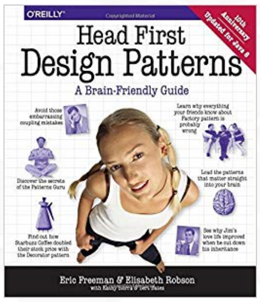
Object-Oriented Software Construction
Second edition

Bertrand Meyer

Prentice Hall, 1997

7

Design patterns (#1/2)



Head First Design Patterns:
A Brain-Friendly Guide

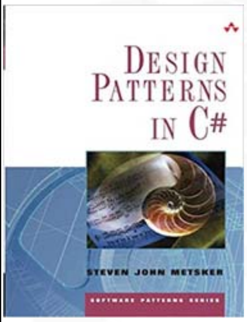
Elisabeth Freeman, Eric Freeman, Bert Bates, Kathy Sierra

O'Reilly Media, 2004

Note: examples are based on Java programming language

8

Design patterns (#2/2)



Design Patterns in C#

Steven John Metsker

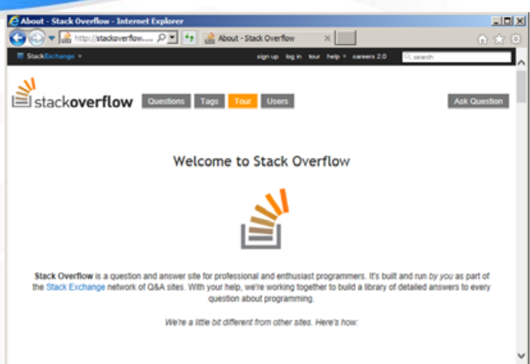
Addison-Wesley, 2004

9

Online resources: http://msdn.microsoft.com

10

Online resources: http://stackoverflow.com



11

Programming paradigms (#1/2)

- non-procedural languages - programs in the form of a sequence of instructions without support for procedures/functions, hence no parameters and value returned by the function (only intrinsic, e.g. SIN, SQR), only global variables, due to memory limitations only 1-2 initial characters of identifiers were significant,
  in the classic BASIC (Dartmouth BASIC, 1964), the code lines were numbered, the control flow was changed by:
  - GOTO <line_number> statement to jump,
  - GOSUB <line_number> to call a subroutine,
  - RETURN to return from a subroutine
- procedural languages - main program + procedures,
  procedures (functions) perform specific services, can be called multiple times, have parameters and can return results: by means of parameters (procedures) or return value (function), they can have local variables,
  e.g. FORTRAN 77 (earlier versions did not support recursion)

12

Programming paradigms (#2/2)

- structural programming - procedures + hierarchy of code blocks, control structures in the form of conditional/selection and loops statements, the main goal of this approach was to eliminate "spaghetti code" (overuse of goto statements) and to support creation of libraries,
  elements of structural programming were introduced into older languages, e.g. FORTRAN, COBOL, BASIC
- object-oriented programming - programming paradigm based on the concept of an object/class reflecting a real world entity,
  the class is a structure consisting of data in the form of fields (attributes) and code in the form of sub-programs related to the class

Short history of Object-oriented programming (#1/2)

- Simula 67 (Kristen Nygaard, Ole-Johan Dahl) - the first fully object-oriented programming language created in 1967 as a result of the development of the previous *Simuli I* project, is an object-oriented extension of the (important) *Algol 60* language, despite its name it is a universal programming language, has an automatic memory reclaim mechanism (garbage collector), supports concurrent programming, however the language has not been commercially successful – it was ahead of requirements,
- Smalltalk – a concept formulated around 1970 by Alan Kay as a development of the *Simula* concept, developed in the 70s and 80s in Xerox laboratories,
  after development versions; Smalltalk-72 and Smalltalk-76 the first publicly available version was Smalltalk-80, it is a combination of *Simula* and *Lisp* (no types) concepts, no distinction between classes and objects (everything is an object), the class is understood as an instance of a higher-class class called a metaclass and thanks to this idea the class hierarchy can embrace the whole system,
- C++ – this programming language was developed by Bjarne Stroustrup in the first half of the 80s, almost completely upwardly compatible with the C language, is a multi-paradigm language, according to many programmers excessively complicated, C++ is not a perfect object-oriented language but appeared at the right time as a transition technology,

Short history of Object-oriented programming (#2/2)

- Java – a language developed by the team working for Sun Microsystems (R.I.P.) in the mid-90s, gained popularity with the avalanche growth of the Internet (independence from architecture), based on previously known concepts but implemented at a new level (unfortunately not all drawbacks of C++ were solved), programs are compiled into bytecode and executed on a virtual machine, supports garbage collection,
- Object Pascal, Objective-C, Ada 95 – examples of incorporating object-oriented extensions to existing structured languages, support multi-paradigm programming,
- C# - an object-oriented programming language designed by a team led by Anders Hejlsberg (Borland Delphi) for Microsoft in 2000, has many common features with other popular object-oriented languages such as Delphi, C ++ and Java, the language is covered by a standard approved by ECMA International (European Computer Manufacturers Association), the leading language of the .NET platform, programs are compiled to the CIL intermediate language executed in the .NET runtime environment (note: the standard allows compilation to native code),

Object-oriented programming paradigm (#1/4)

abstraction - a pillar most often associated with class (less often with a process or method); the model, which in fact does not represent an existing object, is only the basis on which classes are defined (e.g. an abstract data warehouse and not a specific database),

is an abstract "contractor" that performs actions, changes its state and communicates with other objects in the system without revealing how the features have been implemented

Object-oriented programming paradigm (#2/4)

encapsulation – hiding information, hiding implementation details, prevents uncontrolled changes in the object status - internal object data is not available from the outside, changes can only be made using the methods implemented and exposed for this purpose,

Object-oriented programming paradigm (#3/4)

polymorphism – gives the opportunity to vary the functionality of methods with the same interface (name, parameters) depending on the type of object for which it is called,

polymorphism can be, depending on the moment of choosing the appropriate method:

- static - the selection is made during program compilation,
- dynamic (late binding) - the selection is made while the program is running

Object-oriented programming paradigm (#4/4)

inheritance – the ability to use an existing (base, parent) class to create a new (derived, child, subclass) based on the parent one; improves code reuse, the "is-a" relation (A Car is a Vehicle) allows defining more detailed classes on a more general basis; for a specialized class, one does not need to redefine the entire functionality, but only this that is not supported by base class (or is to have a different than provided by parent), inheritance can be:

- single - a class can have only one parent,
- multiple - a class can have more than one parent (which raises some technical problems),

composition – "has-a" relationship (A Car has an engine) class contains other already existing objects, this relationship comes in two variants: in the case of aggregation, the main object and its components can exist independently, in the case of composition (A building has an appartment) means that no component object can exist independently of the main object

the doctrine of *composition over inheritance* advocates implementing "has-a" relationships using composition instead of inheritance

Class concept

Class (object): the basic concept of object-oriented programming.
A class is a template (a declaration of a new type), on the basis of which objects (instances of the class) are created (instantiated).
The class combines data (fields, attributes) and operations (methods, behaviours) to support modelling of real world entities.
The class type defines:

- interface: a set of methods provided by the class, specifying possible behaviour of the object, methods can read and/or modify the state of the object,
- data structure stored in objects: set of fields/attributes, values of all fields are the state of the object,

Objects of a given class share behaviour (= methods) and structure (the same set of fields/attributes) but can have different state (different instance attribute values).
Individual objects of a given class are identifiable and distinguishable (even if they are in the same state).
The class declaration introduces a new type:

```
class Employee
{
  /*
    class body
  */
}
```


Syntax highlighting (colouring) in Visual Studio C#

```
class Employee
{
  private string fullName;
  private double currentSalary;
  public Employee(string newFullName, double newSalary)
  {
    if (newFullName == null)
      throw new ArgumentException("Error: missing employee name");
    this.fullName = newFullName;
    if (newSalary <= 0.0)
      throw new ArgumentException("Error: salary must be positive");
    this.currentSalary = newSalary;
  }
  public string FullName
  {
    get
    {
      return this.fullName;
    }
    set
    {
      if (value == null)
        throw new ArgumentException("Error: missing employee name");
      this.fullName = value;
    }
  }
  public void RaiseSalary(double newSalary)
  {
    if (newSalary <= this.currentSalary)
      throw new ArgumentException("Error: new salary must be higher than the previous one");
    this.currentSalary = newSalary;
  }
}
```

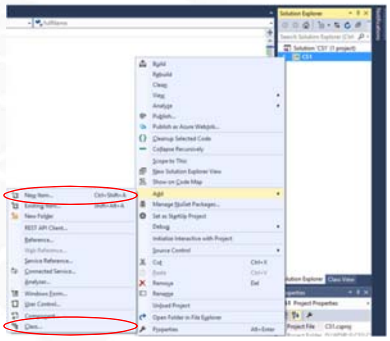
syntax colouring (Microsoft standard shown) improves readability, simplifies code analysis and error search

Adding a class in Visual Studio C# (#1/2)

C# (contrary to i. a. *Ada* and *Java* programming languages does not impose "one class per file" rule) but the recommended convention is to put each class in a separate file with the same name as the class inside.

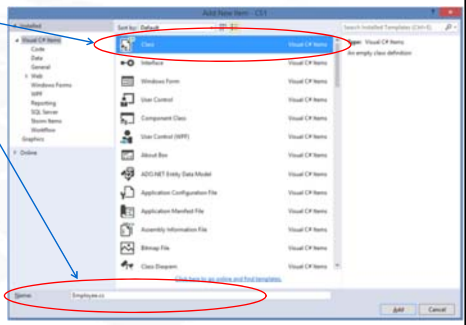
in Visual Studio to (easily) add a new class in a separate file to a project, you can:

- select from the main menu: *Project -> Add class*
- use the keyboard shortcut: *Ctrl + Shift + A*
- right click the project name in *Solution Explorer* window and select *Add ->New item* (or directly *Add -> Class*)



Adding a class in Visual Studio C# (#2/2)

1. select *Class* item
2. enter *Class* name (cs. extension is added automatically)



Class declaration in C# (#1/7)

class members can be declared in any sequence but the following order is recommended:

```
class Employee
{
  private string fullName;
  private double currentSalary;
  public Employee(string newFullName, double newSalary)
  {
    if (newFullName == null)
      throw new ArgumentException("Error: missing employee name");
    this.fullName = newFullName;
    if (newSalary <= 0.0)
      throw new ArgumentException("Error: salary must be positive");
    this.currentSalary = newSalary;
  }
  public string FullName
  {
    get
    {
      return this.fullName;
    }
    set
    {
      if (value == null)
        throw new ArgumentException("Error: missing employee name");
      this.fullName = value;
    }
  }
  public void RaiseSalary(double newSalary)
  {
    if (newSalary <= this.currentSalary)
      throw new ArgumentException("Error: new salary must be higher than the previous one");
    this.currentSalary = newSalary;
  }
}
```

1. data members = fields = attributes

2. constructor(s)
constructor = a method with the same name as the class, invoked automatically when the object is created

3. property(ies)
a property combines field features (can be used in expressions) and method features (automatically launched when getting and setting values)

4. methods

Class declaration in C# (#2/7)

```
class Employee
{
  private string fullName;
  private double currentSalary;
```

1. instance (object) fields (class fields ☞ later, when discussing static components):
each object has its own separate data set (their values form the state of the object),
according to the encapsulation rules, the fields should be private (by default they are, but according to recommeded convention, use the *private* modifier),
modification of fields performer only via methods allows you to check the correctness of the state and react to its change,
private data are not covered by the naming convention but use is *camelCase* recommended

```
    set
    {
      if (value == null)
        throw new ArgumentException("Error: missing employee name");
      this.fullName = value;
    }
  }
  public void RaiseSalary(double newSalary)
  {
    if (newSalary <= this.currentSalary)
      throw new ArgumentException("Error: new salary must be higher than the previous one");
    this.currentSalary = newSalary;
  }
}
```


Class declaration in C# (#3/7)

```
class Employee
{
  private string fullName;
  private double currentSalary;
public Employee(string newFullName, double newSalary)
{
  if (newFullName == null)
    throw new ArgumentException("Error: missing employee name");
  this.fullName = newFullName;
  if (newSalary <= 0.0)
    throw new ArgumentException("Error: salary must be positive");
  this.currentSalary = newSalary;
}
```

2. constructor(s): a method (the procedure ⇒ is does not return the result) with the same name as the class, automatically invoked when the object to initialise its state,
a class can have any number of constructors that differ in parameters (static polymorphism),
only public constructors can be used to create objects outside the class (hierarchy)

```
}
```


Class declaration in C# (#4/7)

```
public string FullName
{
  get
  {
    return this.fullName;
  }
  set
  {
    if (value == null)
      throw new ArgumentException("Error: missing employee name");
    this.fullName = value;
  }
}
```

3. properties: combine the field features - from the point of view of the class user, the property behaves (almost) like a normal field and thanks to that it can be used in expressions (e.g. ++) and at the same time, hidden methods are invoked in the background, each time it is assigned and read (the mutator *set* and the accessor *get* respectively), the mutator and accessor may access to all object members,
the keyword *value* is reserved only if used in the mutator and allows you to reference the value assigned to the property
properties can be public (names compatible with *PascalCase*) or private (*camelCase* recommended)

Class declaration in C# (#5/7)

```
class Employee
{
  private string fullName;
  private double currentSalary;
  public Employee(string newFullName, double newSalary)
```

4. method(s): determine object behaviour - they can read and modify its state, have access to all members of the object/instance,
may be public (names according to *PascalCase*) or private (*camelCase* recommended)

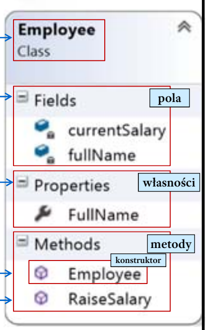
```
public void RaiseSalary(double newSalary)
{
  if (newSalary <= this.currentSalary)
    throw new ArgumentException("Error: new salary must be higher than the previous one");
  this.currentSalary = newSalary;
}
```


Class declaration in C# (#6/7)

UML (Unified Modeling Language) diagram

UML = a standardised modelling language consisting of an integrated set of diagrams

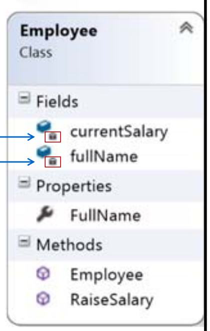
```
class Employee
{
  private string fullName;
  private double currentSalary;
  public Employee(string newFullName, double newSalary)
  {
    if (newFullName == null)
      throw new ArgumentException("Error: missing employee name");
    this.fullName = newFullName;
    if (newSalary <= 0.0)
      throw new ArgumentException("Error: salary must be positive");
    this.currentSalary = newSalary;
  }
  public string FullName
  {
    get
    {
      return this.fullName;
    }
    set
    {
      if (value == null)
        throw new ArgumentException("Error: missing employee name");
      this.fullName = value;
    }
  }
  public void RaiseSalary(double newSalary)
  {
    if (newSalary <= this.currentSalary)
      throw new ArgumentException("Error: new salary must be higher than the previous one");
    this.currentSalary = newSalary;
  }
}
```



Class declaration in C# (#7/7)

```
class Employee
{
  private string fullName;
  private double currentSalary;
  public Employee(string newFullName, double newSalary)
  {
    if (newFullName == null)
      throw new ArgumentException("Error: missing employee name");
    this.fullName = newFullName;
    if (newSalary <= 0.0)
      throw new ArgumentException("Error: salary must be positive");
    this.currentSalary = newSalary;
  }
  public string FullName
  {
    get
    {
      return this.fullName;
    }
    set
    {
      if (value == null)
        throw new ArgumentException("Error: missing employee name");
      this.fullName = value;
    }
  }
  public void RaiseSalary(double newSalary)
  {
    if (newSalary <= this.currentSalary)
      throw new ArgumentException("Error: new salary must be higher than the previous one");
    this.currentSalary = newSalary;
  }
}
```

non-public members are marked with a padlock symbol



Short reminder: value types vs. reference types

- value types are: primitive variables (integer, Boolean, floating-point), enumerative type, structures,
  value types are local copies available using the variable name,
  data is stored on the stack,
- reference types are: strings of characters (string), arrays, classes,
  variable name is a reference (handle, address) to the real data,
  creation of such variable requires allocation of memory from the heap and obtaining a reference to memory area storing data,
  memory allocated to such variables is automatically released when they are no longer used (garbage collector) - not alive = not referenced,
  data is stored on the heap,

Object creation (#1/3)

```
class Employee
{
  // ...
  public Employee(string newFullName, double newSalary)
  // ...
}
```

a constructor

1. step by step:

```
Employee AccountManager;
```

classes are reference types, this statement does not create an object, it only declares a reference to the class object (contrary to C++)

```
AccountManager = new Employee("John Doe", 1000);
```

now an object is created

2. declaration of a reference and creation of an object in single statement:

```
Employee AccountManager = new Employee("John Doe", 1000);
```


Object creation (#2/3)

```
class Employee
{
  // ...
  public Employee(string newFullName, double newSalary)
  // ...
}
```

a constructor

class is a reference type:

```
Employee EmployeeOfTheMonth = new Employee("John Smith", 1000);
/*
  ...
*/
         EmployeeOfTheMonth = new Employee("Jane Public", 2000);
```

- the same reference variable may point to different class objects at different times,
- the memory occupied by objects no longer used in program is automatically reclaimed

Object creation (#3/3)

```
class Employee
{
  // ...
  public Employee(string newFullName, double newSalary)
  // ...
}
```

a constructor

possible ways of object creation depend on public constructors defined in the class:

```
Employee AccountManager;

// AccountManager = new Employee();

AccountManager = new Employee("John Smith", 1000);
```

no default (parameterless) constructor in the class

Object creation failure

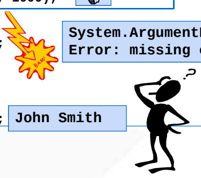
```
class Employee
{
// ...
  public Employee(string newFullName, double newSalary)
  {
   if (newFullName == null)
     throw new ArgumentException("Error: missing employee name");
```

```
Employee Worker;
Worker = new Employee("John Smith", 1000);
try
{
  Worker = new Employee(null, 500);
}
catch(Exception)
{
}
Console.WriteLine(Worker.FullName);
```

Worker = null;

System.ArgumentException:
Error: missing employee name

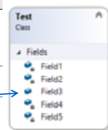
John Smith



Fields: types, initialisation, access

```
class Test
{
  private int Field1;       // explicitly private field, default value = 0
  int Field2 = 5;           // initialised implicitly private field
  public int Field3;        // public field
  private string Field4;    // explicitly private field
  private float Field5;     // explicitly private field
}
```

fields should be *private*!



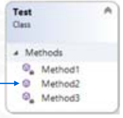
access to fields (encapsulation):

```
Test t1 = new Test();
Console.WriteLine(t1.Field3);
 // remaining fields are private = inaccessible from outside class
```

field directly accessible

Methods: access

```
class Test
{
  private void Method1() { }
  public void Method2() { }
  void Method3() { }
}
```



accessing public methods (encapsulation):

```
Test t1 = new Test();
t1.Method2();
// other methods are explicitly/implicitly private = inaccessible
// outside its class
```


37

Properties: defining, usage

- from the point of view of the user of the object, they look and behave (almost) like plain fields in the object,
- from the implementer's point of view, properties consists of:
  - a private field to store data (for auto-implemented properties automatically created by the compiler)
  - one or two blocks (~methods) of code: mutator *set* automatically invoked when a value is being assigned to the property (often checks the correctness of the assignment), accessor *get* invoked when the value of the property is being read

```
class Test
{
  private int a;      backing field for property A
  public int A
  {
    get
    {
      return a;       read
    }
    set
    { /* ... */
      a = value;      write
    }
  }                   read-write public property
  private int b
  {
    get; // return b;
    set; // b = value;
  }                   auto-implemented private read-write property
}
```


```
Test t1 = new Test();
t1.A = 5;
Console.WriteLine(t1.A);
t1.A++;
Console.WriteLine(t1.A);
```


38

this reference (#1/2)

```
class My2dPoint
{
  private int x, y;
  void MovePoint(int newX, int newY)
  {
    x = newX;
    y = newY;
  }
}
```

```
class My2dPoint
{
  private int x, y;
  void MovePoint(int newX, int newY)
  {
    this.x = newX;
    this.y = newY;
  }
}
```

the "this" keyword is used to refer to the current instance (object) of the class,
Smalltalk, Object Pascal → *self*,
C++, Java → *this*,
Visual Basic → *Me*

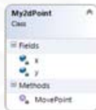
🕯 some coding conventions (implemented in static code analysis tools) recommend using *this* reference even if it is not required, e.g. *StyleCop*, contrary to e.g. *CodeMaid*

39

this reference (#2/2)

```
class My2dPoint
{
  private int x, y;
  void MovePoint(int x, int y)
  {
    this.x = x;
    this.y = y;
  }
}
```

this reference is necessary to differentiate between the method parameter and class field/property if they both have the same name




```
class My2dPoint
{
  My2dPoint Op1(/*...*/)
  {
    /* perform op(s) */
    return this;
  }
}
```

☞ *this* reference may also be useful to access the class instance (object) from outside of it,

☞ returning *this* from methods supports a programming technique called *method chaining* (a *fluent interface*) providing better readability of the source code close to that of ordinary written prose (this API design pattern was first coined by Eric Evans and Martin Fowler), e.g.:

```
StringBuilder s = new StringBuilder()
 .Append("This")
 .Append(" is ")
 .AppendFormat("{0:#.#}", 3.14);
```


40