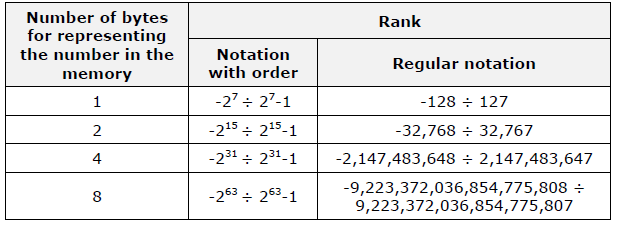
**Representing Negative Numbers:**

* **Range: []**



**Info:**

1. **Straight code** (**signed magnitude**) is the simplest representation of the number. The highest-order bit carries the sign and the rest of the bits hold the absolute value of the number. Here are some examples:
2. The number 3 in signed magnitude is represented as an eight-bit-long number: 00000011.
3. The number -3 in signed magnitude is represented in an eight-bit-long number as: **1**0000011.
4. **Reversed code** (**one’s complement**) is formed from the signed magnitude of the number by inversion (replacing all ones with zeros and vice-versa). Examples:
5. The number -127 in signed magnitude is represented as 1 1111111 and in one’s complement as 1 0000000.
6. The number 3 in signed magnitude is represented as 0 0000011, and in one’s complement looks like 0 1111100.
7. **Additional code** (**two’s complement**) is a number in reversed code to which one is added (through addition). Example:
8. The number -127 is represented with additional code as 1 0000001.

**Info: Binary Coded Decimal**, also known as **BCD** code, in one byte two decimal digits are recorded. This is achieved by encoding a single decimal digit in each half-byte. Numbers presented in this way can be packed, which means that they can be represented in a packed format. If we represent a single decimal digit in one byte we get a non-packed format.

Modern microprocessors use one or several of the discussed codes to present negative numbers, the most widespread method is using two’s complement.

**Integer Types in C#**

**Info:** In C# there are eight integer data types either **signed** or **unsigned**. Depending on the amount of bytes allocated for each type, different value ranges are determined. Here are descriptions of the types:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Size** | **Range** | **Type in .NET Framework** |
| **Sbyte** | 8 bits | -128 ÷ 127 | **System.SByte** |
| **Byte** | 8 bits | 0 ÷ 255 | **System.Byte** |
| **Short** | 16 bits | -32,768 ÷ 32,767 | **System.Int16** |
| **Ushort** | 16 bits | 0 ÷ 65,535 | **System.UInt16** |
| **Int** | 32 bits | -2,147,483,648 ÷ 2,147,483,647 | **System.Int32** |
| **Uint** | 32 bits | 0 ÷ 4,294,967,295 | **System.UInt32** |
| **Long** | 64 bits | –9,223,372,036,854,775,808 ÷  9,223,372,036,854,775,807 | **System.Int64** |
| **Ulong** | 64 bits | 0 ÷ 18,446,744,073,709,551,615 | **System.UInt64** |

**Info:** The most commonly used integer type is **int**. It is represented as a 32-bit number with two’s complement and takes a value in the range **[]**.

**Forcing Overflow Exceptions during Casting**

|  |
| --- |
| **Info:** Sometimes it is convenient, instead of getting the wrong result, when a type overflows during switching from larger to smaller type, to get notification of the problem. This is done by the keyword **checked** which includes a **check for overflow in integer types**:  double d = 5e9d; // 5 \* 10^9  Console.WriteLine(d); // 5000000000  int i = checked((int)d); // System.OverflowException  Console.WriteLine(i); |

|  |
| --- |
| checked  {  int a = int.MaxValue;  a = a + 1;  Console.WriteLine(a);  } |

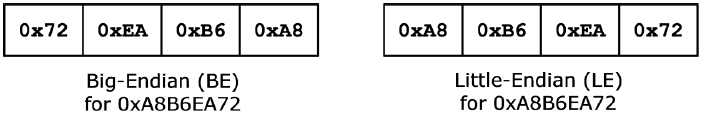
**Big-Endian and Little-Endian Representation**

**Info:** There are two ways for ordering bytes in the memory when representing integers longer than one byte:

**1. Little-Endian (LE)** – bytes are ordered from left to right from the lowest-order to the highest. This representation is used in the Intel x86 and Intel x64 microprocessor architecture.

**2. Big-Endian (BE)** – bytes are ordered from left to right starting with the highest-order and ending with the lowest. This representation is used in the PowerPC, SPARC and ARM microprocessor architecture.

Here is an example: the number **A8B6EA72(16)** is presented in both byte orders in the following way:



**Info:** There are some classes in C# that offer the opportunity to define which order standard to be used. This is important for operations like sending / receiving streams of information over the internet or other types of communication between devices made by different standards. The field **IsLittleEndian** of the **BitConverter** class for example shows what mode the class is working in and how it stores data on the current computer architecture.

**Character Data**

**Encoding Schemes (Encodings)**

**Info:**

**1.** The **ASCII** encoding scheme compares the unique number of the letters from the Latin alphabet and some other symbols and special characters and writes them in a single byte. **(127 characters).**

**2. Windows-1251** encoding scheme compares the unique number of the letters in the Latin alphabet, Cyrillic and some other symbols and specialized characters and writes them in one byte. **(256 characters).**

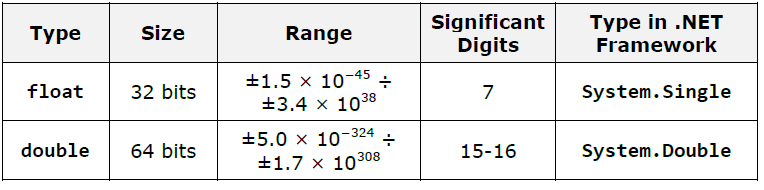
**3.** The **UTF-8** encoding is completely different. All characters in the Unicode standard – the letters and symbols used in all widely spread languages in the world (Cyrillic, Latin, Arabian, Chinese, Japanese, Korean and many other languages and writing systems) – can be encoded in it. **(over half a million symbols)**

In the UTF-8 encoding, the more commonly used symbols are encoded in **1 byte** (Latin letters and digits for example), the second most commonly used symbols are coded in **2 bytes** (Cyrillic letters for example), and the ones that are used even more rarely are coded in **3 or 4 bytes** (like the Chinese, Japanese and Korean alphabet).

1. The **UTF-16** encoding, like UTF-8 can depict text of all commonly used languages and writing systems, described in the Unicode standard. In UTF-16, every symbol is written in **16 bits** (**2 bytes**) and some of the more rarely used symbols are presented as a sequence of two 16-bit values.

**The Float and Double Types in C#**

**Info:** In C# we have at our disposal two types, which can represent floating-point numbers. The **float** type is a 32-bit real number with a floating-point and it is accepted to be called single precision floating-point number. The **double** is a 64-bit real number with a floating-point and it is accepted that it has a double precision floating-point.



**Info:** In C#, **floating-point numbers** literals **by default** are of the **double type**.

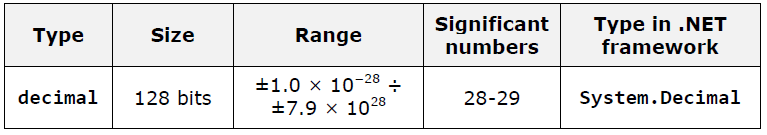
**Info:** **Double** and **Float** types have a field called **Epsilon**, which is a constant, and it contains the smallest value larger than zero, which can be represented by an instance of **System.Single** or **System.Double** respectively.

**-----------------------------------------------------------------------**

**The Decimal Type in C#**

**Info:** The **System.Decimal** type in .NET Framework uses **decimal floating-point arithmetic** and 128-bit precision, which is very suitable for big numbers and

precise financial calculations.



**Info: Decimal** it is tens of times slower than **double** but is irreplaceable for the execution of financial calculations.

**Methods**

**Rules to Name a Method:**

* Capital letter;
* PascalCase;
* Verb **or** verb + noun.

**To name a method it is good to follow these rules:**

- Method name must describe the method’s purpose.

- Method name must begin with capital letter.

**-** The PascalCase rule must be applied.

- The method name must consist of verb, or verb and noun.

**Info:**

**Method** can **NOT be declared inside** the body of another **method.**

A method can be invoked from its own body. Such a call is referred to as **recursion**.

In C# the order of the methods in the class is not important. We are allowed to **invoke (call)** a method before it is declared in code.

**Access Modifier**

**1. Public**

**Info:** It is used to show that method can be called by any C# class, no matter where it is. Public modifiers are not restricted in the meaning of “who” can call them.

**2. Private**

**Info:** Its function is opposite to that of the **public**, i.e. if a method is declared by access modifier **private**, it cannot be called from anywhere, except from the class in which it is declared.

**Info:**

If a method is declared **without an access modifier** (either **public** or **private**), it is **accessible from all classes** in the current assembly, but notaccessible for any other assemblies (let say from other projects in VisualStudio).

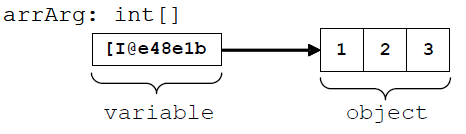
**Info:**

To call a **static** **method** there is **no need** to have **an instance of a class** in which the static method is declared.

**Array**

**Info:**

An array, as any other reference type, consists of a variable-pointer (**object reference**) and a **value** – the real information kept in the computer’s memory (we call it an **object**). In our case the object is the real array of elements. The address of this object, however, is kept in the variable (i.e. the address where the array elements are placed in the memory):

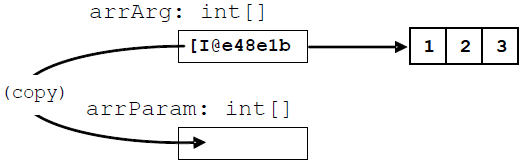
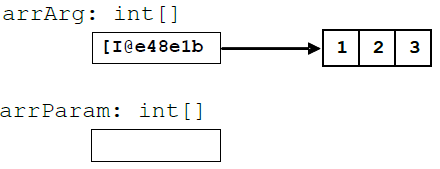
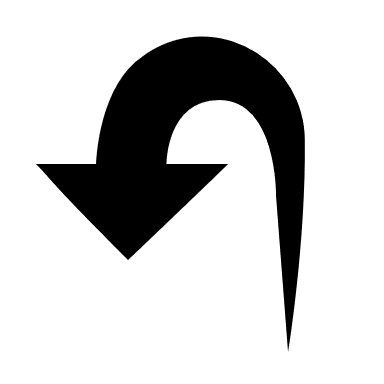


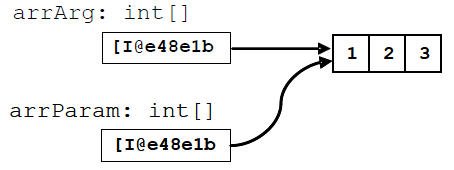
**Info:**

By passing arguments of reference type, only the value of the variable that keeps the address to the object is copied. Note that this **does not copy the object itself**.

**Info:**

By passing the argument that are of reference type, the only thing that is copied is the variable that keeps the reference to the object, but not the object data.





**Info:**

The difference between dealing with arguments of primitive and reference type is in the way they are passed: **primitive types are passed by their values**, **the objects, however, are passed by reference**.

**Methods**

**Variable Number of Arguments (var-args)**

**Info:**

**The last element from the list declaration** – **<params>**, is theone thatallows passing of random count of arguments of type **<var\_type>**, for eachinvocation of the method.In the declaration of that element, before its type **<var\_type>** we must add **params**: "**params <var\_type>[]**". The type **<var\_type>** can be eitherprimitive or by reference.

The element of the parameters list, that allows **passing of variable number** of arguments by invocation of a method, must always be declared at the end of the method’s parameters list.

**Info:**

We can pass a value by a particular **parameter name**, by setting the parameter’s name, followed by a colon and the value of the parameter. An example of using **named arguments** is shown below:

static void Main()

{

// Passing z by name and x by position

SomeMethod(1, z: 3);

// Passing both x and z by name

SomeMethod(x: 1, z: 3);

// Reversing the order of the arguments passed by name

SomeMethod(z: 3, x: 1);

}

**Info:**

The most important aspect of creating an **unambiguous declaration of a method in C#** is the **definition of its signature** and the **type of the method’s parameters** in particular.

**The Operator "return"**

**Info:**

When the method has **void** for returned value type, then after **return**, there would be no expression to be returned. In that case **return** usage is only used to **stop the method’s execution**:

static void PrintPositiveNumber(int number)

{

if (number <= 0)

{

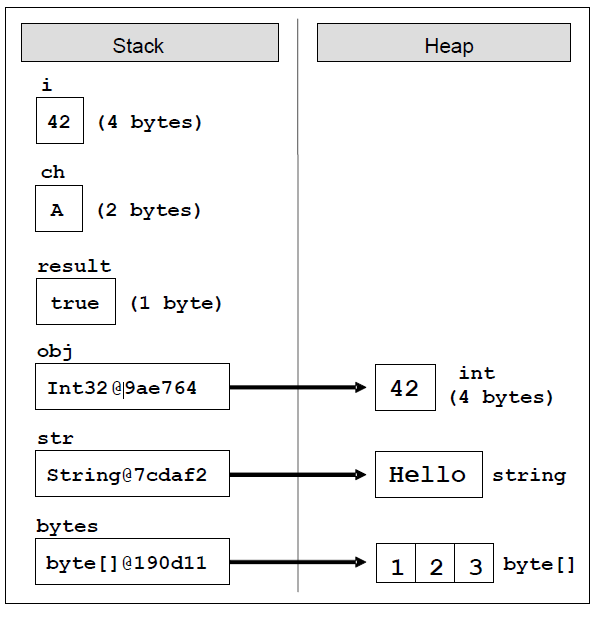
// If the number is NOT positive, terminate the method

return;

}

Console.WriteLine(number);

}



**"Selection sort" Algorithm**

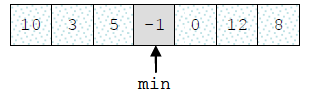
**Info:**

This method considers the array as two parts – sorted and unsorted. The sorted part is in the left side of the array, while the unsorted is in the right. For each step of the algorithm, the sorted part expands to the right with one element and the unsorted shrinks with one element from its left part.

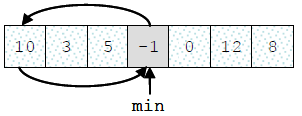
Let’s take a look at an example. So assume we have the following unsorted array and we want to order its elements by **selection sorting**:



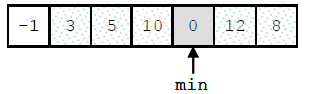
On each step our algorithms must find the minimal element in the unsorted part of the array:



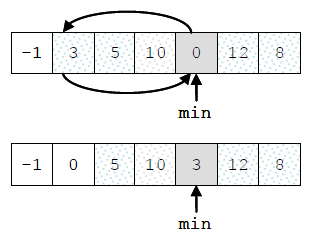
Then the minimal element must swap with the first element from the unsorted part of the array:



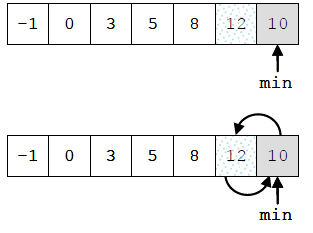
Then we look for the minimal element again, from the rest of the unsorted part of the array (all elements except the first one):



That minimal element now exchanges with the first from the unsorted part:



So this step is repeated until the unsorted part of the array reaches a length of 0, i.e. it is empty:





As a result the array is **sorted**:

**Recursion**

**Info:**

We call an object **recursive** if it contains itself, or if it is defined by itself.

**Recursion** is a programming technique in which **a method makes a call to itself** to solve a particular problem. Such methods are called **recursive**.

If by using **recursion** we reach a **simpler**, **shorter** and **easier** **for understanding solution**, not causing **inefficiency** and other **side effects**, then we can prefer **recursive solution**. **Otherwise**, it is better to think of **iteration**.

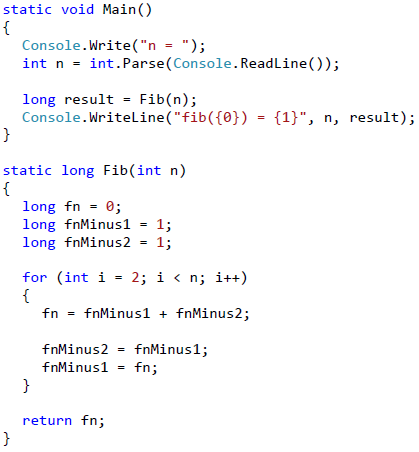
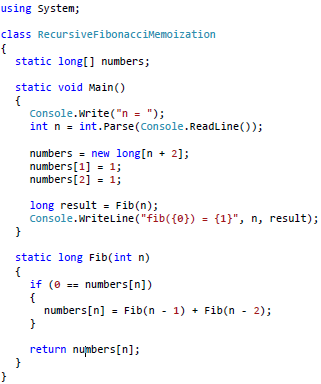
We can **optimize the recursive method** for calculating the Fibonacci numbers by remembering (saving) the already calculated numbers in an array and making recursive call only if the number we are trying to calculate has not been calculated yet.

Thanks to this small **optimization technique** (also

known in computer science and dynamic optimization as **memoization** (not to be confused with **memorization**) the recursive solution would work for linear count of steps.

**Avoid recursion**, unless you are certain about how it works and **what has to happen behind the scenes.** Recursion is a great and powerful weapon, with which you can easily shootyourself in the leg. **Use it carefully!**

**Fibonacci Memoization vs Fibonacci Iterative Solution**



**Recursion**

**Info:**

Ordinarily each recursion could **boil down to iteration by using a stack** of the calls (which is created through program execution), but this is complicated and there is no benefit from doing this.

Use **recursion** for **branched recursive calculations** (and ensure **each value is calculated only once**). For **linear recursive calculations** prefer using **iteration.**

**OOP**

**Info:**

We are going to mention only thatprogramming techniques of OOP often include **encapsulation**, **abstraction**, **polymorphism** and **inheritance**.

Examples of abstract objects are the data structures stack, queue, list and tree.

* **States** – these are the characteristics of the object which define it in a way and describe it in general or in a specific moment.
* **Behavior** – these are the specific distinctive actions, which can be done by the object.
* **Data members** – embedded in objects variables, which describe their states.
* **Methods** – we have already considered them in details. They are a tool for building the objects.

**Info:**

Furthermore, each object is an **instance** of exactly one specific class.

**Classes** in C# can contain the following elements:

* **Fields** – member-variables from a certain type;
* **Properties** – these are a special type of elements, which extend the functionality of the fields by giving the ability of extra data management when extracting and recording it in the class fields.
* **Methods** – they implement the manipulation of the data.

**Info:**

It is important to know that the implementation of the logic in classes is **encapsulated** (hidden) inside them.

For the programmer it is important what they do, not how they do it and for this reason a great part of the classes is not publicly available (**public**). With system classes the implementation is often not available at all to the programmer.

Thus, new **layers of abstraction** are created which is one of the basic principles in OOP.

**Creating and Releasing Objects**

**Info:**

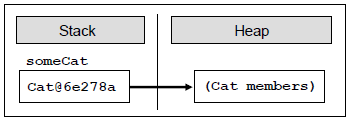
The creation of objects from preliminarily defined classes during program execution is performed by the **operator new**.

The newly created object is usually assigned to the variable from type coinciding with the class of the object (this, however, is not mandatory).

We are going to note that in this assignment the object is not copied, and only a **reference** to the newly created object is recorded in the variable (its address in the memory). Here is a simple example of how it works:

Cat someCat = new Cat();

The variable **someCat** of type **Cat** we assign the newly created **instance** of the class **Cat**. The variable **someCat** remains in the **stack**, and its value (the instance of the class **Cat)** remains in the **managed heap**:



**Creating Objects with Set Parameters**

**Info:**

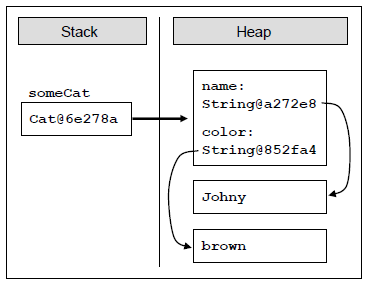
Cat someCat = new Cat(“Johnny”, “brown”);

When creating an object with the operator **new**, two things happen: memory is set aside for this object and its data members are initialized.

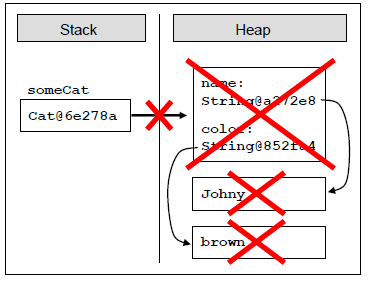
The **initialization** is performed by a special method called **constructor**. In the example above the initializing parameters are actually parameters of the constructor of the class.

As the member variables **name** and **color** of the class **Cat** are of reference type (of the class **String**), they are also recorded in the **dynamic memory (heap)** and in the object itself are kept their references (addresses / pointers).

The following figure illustrates how the **Cat** object is represented in the computer memory (arrows illustrated the **references** from one object to another):



Objects to which there is no reference in the program at certain moment are **automatically released** and the memory they take up is released.





someCat = null;

This does not destroy the object immediately,

but puts it in a state in which it is inaccessible

to the program and the next time the garbage

collector cleans the memory it is going to be

released.

**Constructors**

**Info:**

The **constructor** is a special method of the class, which is called automatically when **creating an object** of this class, and performs initialization of its data (**this is its purpose**).

The constructor has no type of returned value and its name is not random, and mandatorily coincides with the class name.

The constructor can be **with or without parameters**. A constructor without parameters is also called **parameterless constructor**.

**Static Fields and Methods**

**Info:**

**A static field or method** in a given class is defined with the keyword static, placed before the type of the field or the type of returned value of the method.

**When to Use Static Fields and Methods?**

**Info:**

Let’s interpret the **class as a category of objects**, and **the object as a representative of this category**.

Then the **static members** reflect the state and the behavior of the **category itself**, and the **non-static** the state and the behavior of the **separate representatives of the category**.

Now we are going to pay special attention to the **initialization of static and non-static fields**.

**Info:**

**Static fields are initialized when the data type (the class) is used for the first time, during the execution of the program.**

**A class that has only private constructors cannot be instantiated. Such class usually has only static members and is called "utility class".**

**Constants**

**Info:**

**Constants** in C# are immutable variables whose values are assigned during their initialization in the source code of the program and after that they cannot be changed.

In C# constants are written in Pascal Case (the words in the name, merged together, each of them starts with an uppercase letter, and the rest of them are lowercase).

**Namespaces**

**Info:**

When naming namespaces in the hierarchy we use the character **.** as a separator (dot notation). For example, the namespace **System** from .NET Framework contains in itself the sub-namespace **Collections** and thus the full name of the nested namespace **Collections** is **System.Collections**.

**Classes are required to have unique names only within the namespaces, in which they are defined.**

**Inclusion of namespaces is not recursive, i.e. when including a namespace the classes from the nested namespaces are not included.**

**Important MAXIM!!**

**Info:**

When we write a program, we describe step-by-step what the computer must do (at least in **imperative programming**; in the **functional programming** things look a bit different) and in most of the cases we rely that the program will execute normally.

Indeed, most of the time, programs are following this normal pattern, but there are some exceptions. Let’s say we want to read a file and display its contents on the screen. Let’s assume the file is located on a remote server and during the process of reading it, the connection goes down. The file then will be only partially loaded. The program will not be able

to execute normally and show file’s contents on the screen. In this case, we have an **exception** from the normal (and correct) program execution and this exception must be reported to the user and/or the administrator.

**Exceptions**

**Info:**

Exceptions are one of the main paradigms of object-oriented programming (OOP)!

Another fundamental concept is **exceptions hierarchy**. In OOP, exceptions are classes and they can be **inherited to build hierarchies**.

In OOP, it is recommended to **use exceptions for managing error situations or unexpected events** that may arise during a programexecution.

Sometimes exceptions are used not so much to signal a problem but to handle some expected event. This is not considered a good practice as exceptions should not control the normal flow of the program.

**Exceptions in .NET**

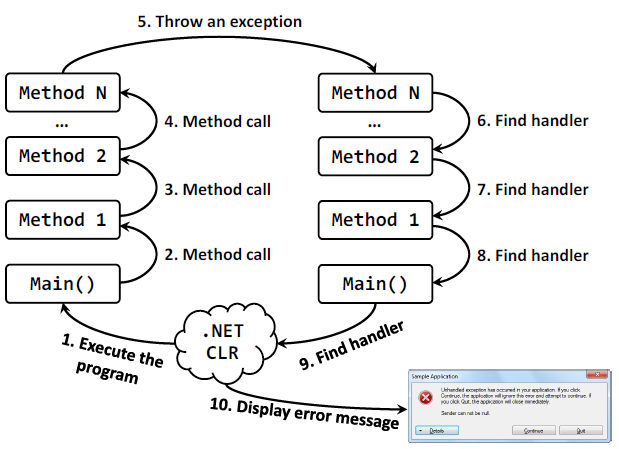
**Info:**

**Exception** in .NET is an **object**, which signals an error or an event, which is not anticipated in the normal program flow.

Each exception in .NET contains the so-called **stack trace**, which gives information of where exactly the error occurred.

The general principle is that when a new method is called, it is **pushed on top of the stack**. When the method finishes, it is **pulled back from the stack**. At any given point in time, the call-stack contains all the methods called during the execution – from the starting method **Main(…)** to the last called method, which is currently executing, along with their local variables and arguments taken as input.

The **exception handling mechanism** follows a reversed process. When an exception is thrown, CLR begins searching an **exception handler** in the call-stack starting from the method that has thrown the exception. This is repeated for each of the methods down the call-stack until a handler is found which catches the exception. If **Main(…)** is reached and no handler is found, CLR catches the exception and usually displays an error message (either in the console or in a special error dialog box).

The described **method call and exception handling process** could be visualized in the following diagram (steps 1 through 5):

**Try-catch**

**Info:**

The **try-catch** construct consists of one **try** block and one or more **catch** blocks.

Showing the full information about the exception to the **end user** is **not always a good practice**!

**Stack Trace:**

The **stack trace** contains **detailed information about the exception** including where exactly it occurred in the program. During debugging the stack trace is a priceless tool.

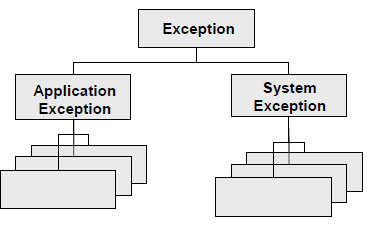
The debug information is not included in the .NET assemblies but is in separate files called 'debug symbols' (**.pdb**).

**Exceptions Hierarchy**

**Info:**

There are two types of exceptions in .NET Framework:

* **ApplicationException** - exceptions thrown by the applications we develop;
* **SystemException** - exceptions thrown by the runtime.

Each of these is a base class for a hierarchy of exception classes:

The **Exception** class contains a copy of the call-stack at the time the exception instance was created. The class also has a (usually) short message describing the error (filled in by the method throwing the exception). Every exception could have a **nested exception** also sometimes called an **inner exception**, **wrapped exception** or **internal exception**.

The ability to wrap an exception with another exception is very useful in some cases and allows exceptions to be linked in the so called **exception chain**.

When we, as developers, design our own exception classes, it is a good practice to inherit from **ApplicationException** and not directly from **SystemException** (or even worse – directly from **Exception**). **SystemException** should only be inherited internally within the .NET Framework.

**Exceptions**

**Info:**

**A method should only handle exceptions which it expects and which it knows how to process. All the other exceptions must be left to the calling method.**

Handling exceptions at different levels allows the error conditions to be handled at the most suitable place for the particular error. This allows the program code to be clear and structured and the flexibility achieved is enormous.

**The code in the finally block will not be executed if while executing the try block, CLR is unexpectedly terminated, e.g. if we stop the program through Windows Task Manager.**

The **finally** block is priceless when we need to free an external resource or make any other cleanup. The **finally** block guarantees that the cleanup operations will not be accidentally skipped because of an unexpected exception or because of execution of **return**, **continue** or **break**.

**The Keyword “using”**

**Info:**

using (StreamReader reader = new StreamReader(fileName))

{

// Use the reader here

}

The above **simplified form of the "dispose pattern"** is simple to write, simple to use and simple to read and is guaranteed to release correctly the allocated resources specified in the brackets of the **using** statement.

Use the **using statement** with **all classes that implement the IDisposable interface**.

**Exceptions**

**Info:**

Catching **Exception** and all of its inheritors is not a good practice. It is better to catch more specific groups of exceptions like **IOException** or just one type of exception like for example **FileNotFoundException**.

It is not a good practice to rely on exceptions for expected events for another reason: performance. Throwing an exception is **time consuming operation**. An object has to be created to hold the exception, the stack trace has to be initialized and handler for this exception has to be found and so on.

**It is hard to define the exact border between expected and unexpected. In general expected event is something related to the program functionality. Input of wrong file name for example. Power cut during the execution of the program, from the other hand, is unexpected event.**

**Throw Exceptions at the Appropriate Level of Abstraction!**

**Each exception should carry detailed information about the problem.**

**Always give adequate, detailed and correct error message when throwing exceptions! The user of your code should be able to tell what and where is the problem and what caused it when reading the error message.**

**Be careful not to show messages with incorrect content!**

**A method should either do the work it is created for or throw an exception. Any other behavior is incorrect!**

**A method should either do the work it is created for or throw an exception. In case of wrong input the method should throw an exception and should not return a wrong result!**

**A very common mistake with exceptions is to catch all exceptions no matter what type they are.**

**We should handle only errors that we expect and we are prepared for. We should leave the other errors (exceptions) so they are caught by another method that knows how to handle them.**

**A method should not catch all exceptions – it should only catch the ones it can process correctly.**

**Strings**

**Info:**

The type **string** is more special from other data types. It is a class and as such it complies with the principles of object-oriented programming. Its values are stored in the dynamic memory (**managed heap**), and the variables of type **string** keeps a **reference** to an object in the heap.

The **string** class has an important feature – the character sequences stored in a variable of the class are never changing (**immutable**).

After being assigned once, the content of the variable does not change directly – if we try to change the value, it will be saved to a new location in the dynamic memory and the variable will point to it.

Displaying special characters in the source code is called **escaping**. This is not equivalent to setting a variable and allocating memory for it!

With the declaration we inform the compiler that the variable **str** will be used and the expected type for it is **string**. We do not create a variable in the memory and it is not available for processing yet (value is **null**, which means no value).

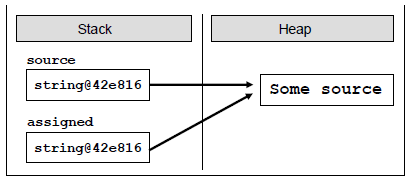
Before setting a specific value to the string, its value is **null**. Uuninitialized variables of type **string** do not contain empty values, it contains the special value **null** – and each attempt for manipulating such a string will generate an error (exception for access to a missing value **NullReferenceException**).

**Assigning Value of Another String**

string source = "Some source";

string assigned = source;

First, we declare and initialize the variable **source**. Then the variable **assigned** takes the value of **source**. Since the **string** class is a **reference type**, the text "**Some source**" is stored in the dynamic memory (heap) on an address defined by the first variable.



**When you want just to consider whether the values of two strings are equal or not, please use the method Equals(…) or the operator ==. The methods CompareTo(…) and string. Compare(…) are designed to be used when the lexicographical order is needed.**

**Strings are immutable! Any change of a variable of type string creates a new string in which the result is stored. Therefore, operations that apply to strings return as a result a reference to the result.**

**Remember that in C# indexing into strings start from 0.**

The **methods IndexOf(…)** and **LastIndexOf(…)** search the contents of the text sequence, but in a different direction. The search with the first method starts from the beginning of the string towards the end, while the second method – the search is done backwards. If we are interested in the first encountered match, then we use **IndexOf(…).** If we want to search the string from its end (for example to detect the last dot in a file name or the last slash in an URL address), then we use **LastIndexOf(…).**

int index = str.IndexOf("C#"); // index = 0

index = str.IndexOf("Course"); // index = 15

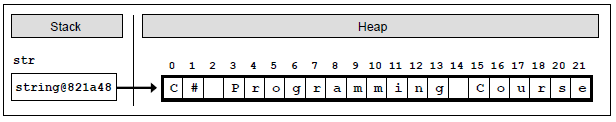
index = str.IndexOf("COURSE"); // index = -1

index = str.IndexOf("ram"); // index = 7

index = str.IndexOf("r"); // index = 4

index = str.IndexOf("r", 5); // index = 7

index = str.IndexOf("r", 10); // index = 18

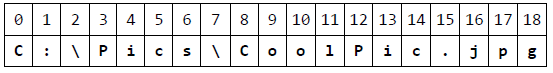


string path = "C:\\Pics\\CoolPic.jpg";

string fileName = path.Substring(8, 7);

// fileName = "CoolPic"

**Calling the method Substring(startIndex, length), extracts a substring from a string, which is located between startIndex and (startIndex + length – 1) inclusively. The character at the position startIndex + length is not taken into consideration!** **For example, if we point Substring(8, 3), the characters** **between index 8 and 10 inclusively will be extracted.**



When splitting strings and adding as a second parameter the constant **StringSplitOptions.RemoveEmptyEntries** we instruct the method **Split(…)** to work in the following way: “Return all substrings from the variable that are split by given list of separators. If you meet two or more neighboring separators, consider them as one.”

**Please note that we must list all the characters we want to eliminate, including the empty spaces (spaces, tabs, new line, etc.). Without a ' ' in the array trimChars, we would not get the desired result!**

If we want to remove the white spaces only at the beginning or in end of the string, we can use the methods **TrimStart(…)** and **TrimEnd(…).**

**Constructing Strings: the StringBuilder Class**

**Info:**

Strings in C# are **immutable.** This means that any adjustments applied to an existing string do not change it but return a new string. For example, using methods like **Replace(…)**, **ToUpper(…)**, **Trim(…)** do not change the string, which they are called for. They allocate **a new area in** **the memory** where the new content is saved. This behavior has many advantages but in some cases can cause performance problems.

**Strings Concatenation in a Loop: Never Do This!**

Allocating a new area, recording a value, creating a new variable and referencing it in the memory is timeconsuming process that would be a problem when repeated many times, typically inside a loop.

**Unlike** other programming languages, in C# **is not necessary to manually dispose the objects stored in memory.** There is a special mechanism called a **garbage collector (memory cleaning system),** which takes care of clearing the unused memory and resources.

The garbage collector is responsible for disposing of objects in dynamic memory when they are no longer used.

**Much more elegant and appropriate way to concatenate strings in a Loop is using the StringBuilder class. Let’s see how it works.**

**StringBuilder** is a class that serves to build and change strings.

The **StringBuilder class** is an implementation of a string in C#, but different than the **class String.**

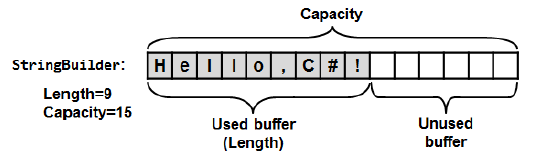
**StringBuilder keeps a buffer with a certain capacity** (16 characters by default). The buffer is implemented as an array of characters that is provided to the developer by a user-friendly interface – methods that quickly and easily add and edit elements of the string.

Once the internal buffer of the **StringBuilder** is full, it automatically is doubled (the internal buffer is resized to increase its capacity while its content is kept unchanged). **Resizing is a slow operation** but is happens rarely so the total performance is good.

StringBuilder sb = new StringBuilder(15);

sb.Append("Hello, C#!");

After creating the object and storing the value in it, the **StringBuilder** will look as follows:



Normally, adding a new character to

the variable does **not create a new**

**object in the memory** but **use the**

**already allocated and unused**

**space.**

**Info:**

One of the interesting concepts in .NET is that practically **every object of a class and primitive variables** can be **presented as text**. This is done by the method **ToString(…)**, which is present in all .NET objects. It is implicit in the definition of the **object** class – the base class that all .NET data types inherit directly or indirectly.

The default implementation of the **ToString(…)** method in the **object** class returns the full name of the class. All classes that do not explicitly redefine the behavior of the **ToString(…)** are using this implementation.

Most classes in C# have their own implementation of the method, which represents readable and understandable content in text form.

**Parsing of data**

**Data parsing** means to obtain a value of a given type from the text representation of this value in a specific format, i.e. **converting from text to** **some other data type**, the opposite of **ToString()**.

**When converting types, we should not rely only on trusting the user. Always check the correctness of the input user data! Otherwise there could be an exception that could change the normal program logic.**

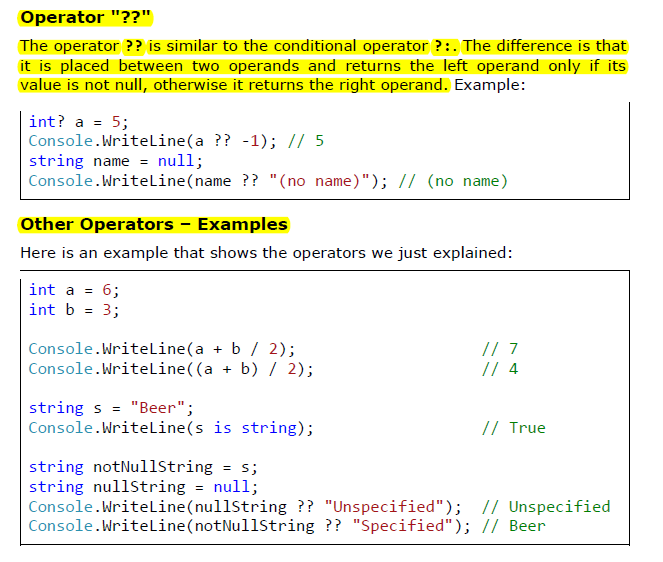
**Defining Classes**

**Info:**

**Class** in the **OOP** is called a definition (**specification**) of a given type of objects from the real-world. The class represents a pattern, which describes

the different states and behavior of the certain objects (the copies), which are created from this class (pattern).

**Object** is a copy created from the definition (**specification**) of a given **class**, also called an **instance**. When one object is created by the description of one class we say **the object is from type "name of the class".**



**Modifiers and Access Levels (Visibility)**

**Info:**

A **modifier** is a reserved word and with the help of it we add additional information for the compiler and the code related to the modifier.

In C# there are four **access modifiers**:

* **public**
* **private**
* **protected**
* **internal**

The access modifiers can be used only in front the following elements of the class: **class declaration**, **fields**, **properties** and **methods**.

The levels of access in .NET are:

* **public**
* **protected**
* **internal**
* **protected internal**
* **private**

**Access Level "public"**

**Info:**

When use modifier **public** that this element can be accessed from **every class**, no matter from **the current project (assembly)**, from **the** **current namespace**. The access level **public** defines the miss of restrictions regarding the visibility. This access level is the least restricted access level in C#.

**Access Level "private"**

**Info:**

The access level **private** is the one, which defines **the most restrictive** **level of visibility** of the class and its elements. The modifier **private** is used to indicate, that the element, to which is issued, **cannot be accessed from** **any other class** (except the class, in which it is defined), even if this class exists in the same **namespace**.

**This is the default access level**, i.e. it is used when there is no access level modifier in front of the respective element of a class (**this is true only for elements inside a class**).

If we want to be exhaustive, we have to mention that as access modifier for a class can be used the visibility modifier **private**, but this is related to the term "**inner class**" (**nested class**). Private classes like other private members are accessible only inside the class which defined them.

**Access Level "internal"**

**Info:**

The modifier **internal** is used to limit the access to the elements of the class only to files **from the same assembly**, i.e. the same project in Visual Studio.

When we create several projects in Visual Studio, the classes from will be compiled in different assemblies.

If we declare one class with access modifier **internal**, one will be **accessible only from the same namespace**. It means that only the classes from thesame assembly can create objects from this type class and to have access tothe methods and fields (with related access level) of the class.

This accesslevel is **the default**, where it is not used access modifier by the declaration ofthe class.

**Assembly**

**Info:**

.NET assemblies are **collections of compiled types** (classes and other types) and **resources**, which form a logical unit.

Assemblies are stored in a binary file of type **.exe** or **.dll**. All types in C# and as general in .NET Framework can reside only inside assemblies. By every compilation of a .NET application one or several assemblies are created by the C# compiler and each assembly is stored inside an **.exe** or **.dll** file

The access levels, which an outer class can have, are only **public** and **internal**. Inner classes can be defined with other access levels.

**The Reserved Word "this"**

**Info:**

The reserved word **this** in C# is used to **reference the current object**, when one is used from method in the same class. \

**This is the object**, which method or constructor is called. The reserved word can be deemed as an **address** (**reference**), given priory from the language authors, with which we access the elements (fields, methods, constructor) of the own class.



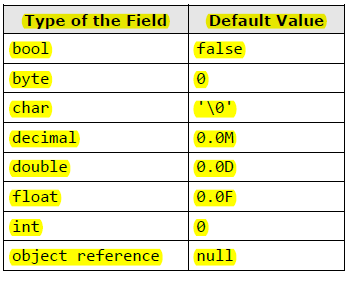
**Default Values of the Fields**

**Info:**

Every time, when we create **a new object of a given class**, it is **allocated memory in the heap** for **every field from the class.**

In order this to be done the memory is **initialized automatically with the default values** for the certain field. The fields, which do not have explicitly a default value in the code, use the default value specified for the .NET type, to which they belong.

**When an object is created all of the fields are initialized with their respective default values in .NET, except if they are not explicitly initialized with some other value.**



The value of all types is 0 or

something similar.

For the most used types these

values are as the follows:

**Unlike fields, local variables are not initialized with default values when they are declared!**

**A good programming practice is**: when we declare fields in the class, **to explicitly initialize them with some default value**, even if the default value is zero. This will make our code clearer and easy to read.

**Modifiers "const" and "readonly"**

**Info:**

In the declaration of one field is allowed to use the modifications **const** and **readonly**. The fields, declared as **const** or **readonly** are called **constants**.

They are used when a certain **value is used several times**. These values are declared only ones without repetitions.

Examples of constants in the .NET Framework are the mathematical constants **Math.PI** and **Math.E**, and as well the constants **String.Empty** and **Int32.MaxValue**.

**Constants Based on "const"**

**Info:**

The fields, declared with **const**, have to be initialized during the de facto declaration and afterwards **theirs value cannot be changed**.

**They can be accessed without to create an instance (an object) of the class and they are common for all created objects in our program!**

Something more, when we compile the code, the places where **const** fields are referred are replaced with theirs particular values directly without to use the constant variable at all.

For this reason the const fields are called **compile-time constants,** because they are replaced with the value during the compilation process.

**Constants Based on "readonly"**

**Info:**

The modifier **readonly** creates fields, which values cannot be changed once they are assigned. Fields, declared as **readonly**, allow one-time initialization either in the moment of the declaration or in the class constructors.

Later theirs values cannot be changed. Because of this reason, the **readonly** fields are called **run-time constants** – constants, because their values cannot be changed after assignment and run-time, because this process happens during the execution of the program (in runtime).

**The access to the non-static elements of a class (fields and methods) is done via the reserved word this and the operator for access – "dot".**

**Skip "this" Keyword When Accessing Non-Static Data**

**Info:**

When we access the fields of a class or we call its non-static methods, it is possible to **omit the reserved word this**.

When it is not required explicitly the reserved word this can be skipped when we access the elements of the class. For better readability use this keyword even when not required.

**Hiding Fields with Local Variables**

**Info:**

This is so, because C# allows defining local variables, which names match with fields of the class. If this happens, we say that the scope of the local variable overlays the field variable (**scope overlapping**).

**Visibility of Fields and Methods**

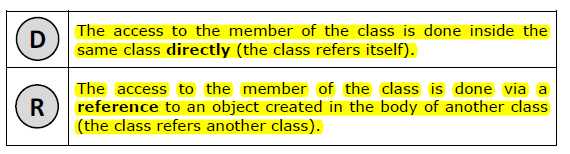
**Info:**

**If two classes are not visible one to other, then their members (fields and methods) are not visible also, regardless of what kind of access levels their elements have.**

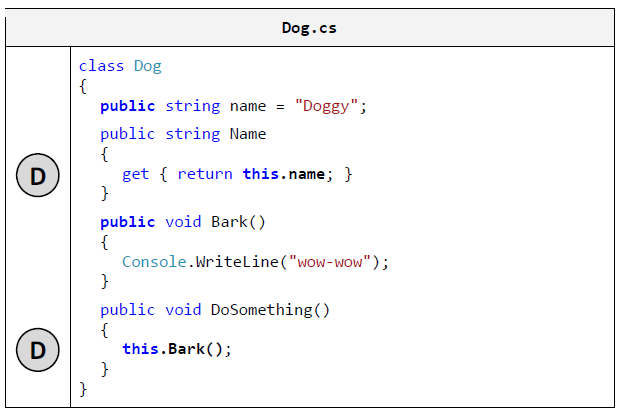
**Access Level "public"**

**Info:**

When a method or a value of a class is declared with access level **public**, the last **can be used from other classes**, independently from the fact if another class is declared in the same namespace, assembly or outside of it.



When the members of both classes are **public**, we have the following:

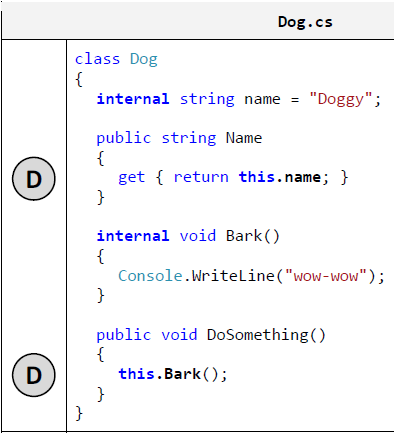




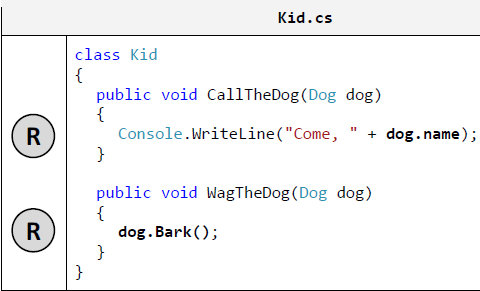
**Access Level "internal"**

**Info:**

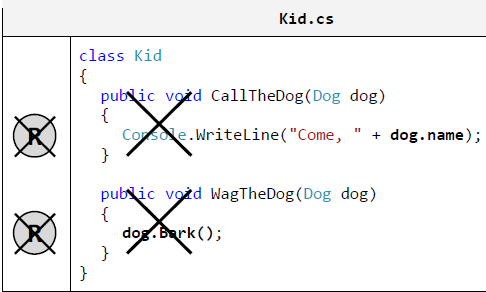
When a member of some class is declared with access level **internal**, then this element from the class **can be accessed from every class in the same assembly** (i.e. in the same project in Visual Studio), but not from classes outside it (i.e. from other projects in Visual Studio – from the same solution or from a different solution).



Respectively, for the class **Kid**, we discuss two cases: - When the class in **the same assembly**, then the access to the elements of **Dog** will be allowed, independent of whether the classes are in the same namespace or not:



- When the class **Kid** is **external for the assembly**, in which **Dog** is declared, then the access to the field **name** and the method **Bark()** will be denied:



Actually the access level **internal** for members of the class **Dog** is impossible for two reasons:

* insufficient visibility of the class;
* insufficient visibility of its members.

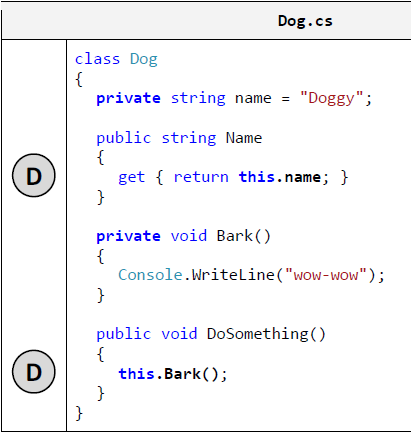
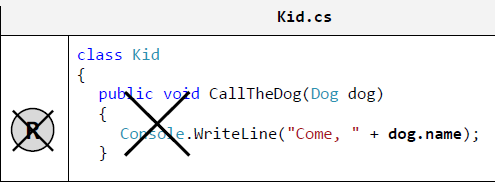
**Access Level "private"**

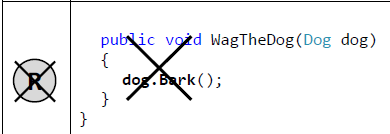
**Info:**

The access level, which is **the most restrictive**, is **private**. The elements of the class, which are declared with access modifier **private** (or without any, because **private** is the default one), **cannot be accessed outside of the class** in which they are declared.

Therefore, if we declare the field **name** and the method **Bark()** of the class **Dog** with access modifier **private**, there is no problem to access them from the same instance of the class **Dog**, but access from any other classes is not permitted.

Below is the figure about the access level **private**:





**Info:**

We should know, when we assign access modifier to a filed, one in most of the cases has to be **private**, because this ensures the highest level of security applied to the field.

Respectively, the access and the modification of the value from other classes (if it is required) **will be done only via properties or methods.**

**Constructors**

**Info:**

In object-oriented programming, when creating an object from a given class, it is necessary to call a special method of the class known as a **constructor**.

**What Is a Constructor?**

**Info:**

**Constructor** of a class is a pseudo-method, which does not have a return type, has the name of the class and is **called using the keyword new**.

The task of the constructor is to initialize the memory, allocated for the object, where its fields will be stored (those which are not **static** ones).

**Calling a Constructor**

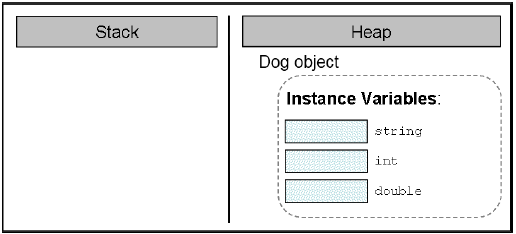
**Info:**

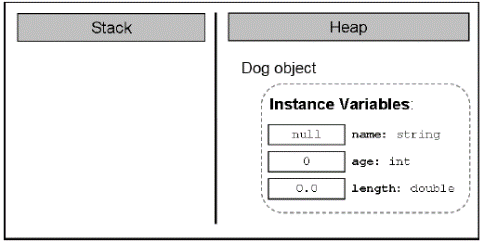
The only one way to **call a constructor** in C# is through the **keyword new**. It allocates memory for the new object (in the stack or in the heap, depending on whether the object is a value type or a reference type), resets its fields to zero, calls their constructors (or chain of constructors, formed in succession), **and at the end returns a reference to the newly created object.**

**When it comes to classes they are allocated in the dynamic memory (in the so called "managed heap").**

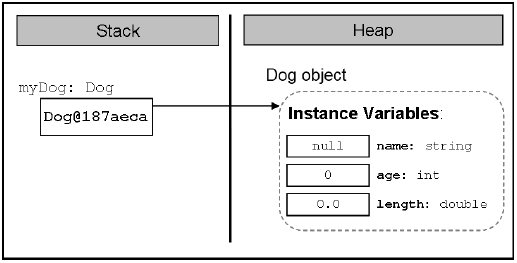
Let’s follow the process of calling a constructor during the creation of new object step by step:

1. First, **memory is allocated** for the object:



2. Next, its **fields (if any) are initialized with the default values** for their respective types:

3. If the creation of the new object is successfully completed, the **constructor returns a reference** to it, which is assigned to the variable **myDog**, from class type **Dog**:



As we already know, the constructors are similar to methods, but they **do not have a return type** (therefore we called them pseudo-methods).

**We should know that only** const **is not allowed to be used in constructors.**

When **creating a new object and calling its constructor**, a new memory is allocated for the non-static fields of the object of the class and they are **initialized with the default values** for their types.

Furthermore, through the constructors we mainly initialize the fields of the class with values set by us and not with the default ones.

string name = "Sharo";

Instead of doing this during the declaration of the field, a better programming style is to assign its value in the constructor:

public class Dog

{

private string name;

public Dog()

{

this.name = "Sharo";

}

// … The rest of the class body …

}

Although we initialize the fields in the constructor, some people recommend **explicitly assigning their type’s default values** during initialization with the purpose of improving the readability of the code, but it is a matter of personal choice:

public class Dog

{

private string name **= null**;

public Dog()

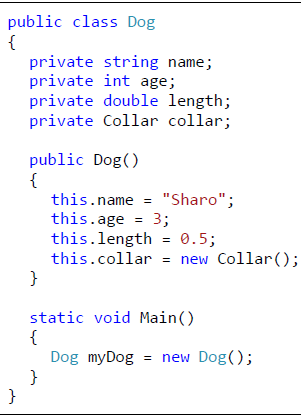
{

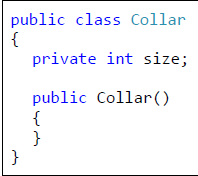
this.name = "Sharo";

}

// … The rest of the class body …

}



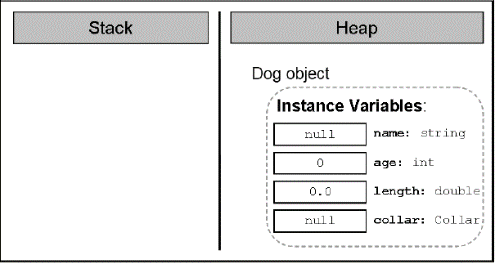


**Representation in the Memory**

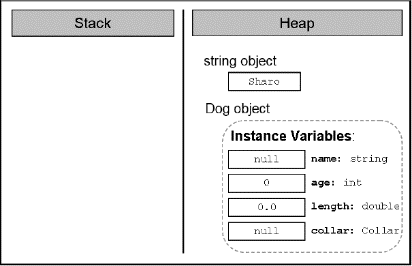
**Info:**

Let’s follow the steps through which the constructor goes, after being called in the **Main()** method.

1. As we know, as a first step it will **allocate memory in the heap** for all the fields and will initialize them with their default values:



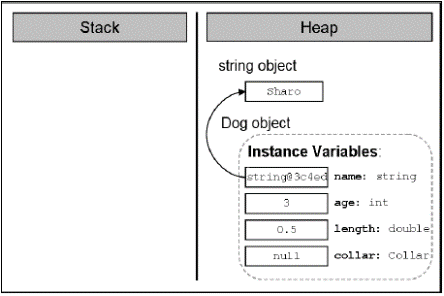
2. Then, the constructor will have to ensure the creation of the object for the field **name**. It will **call the constructor of the class string**, which will do the work on the string creation):



3. Now the constructor will keep

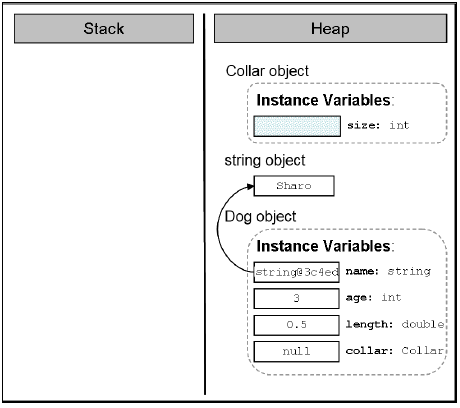
the reference to the new string in

the field **name** of the **Dog** object:



4. Then is the creation of the object from type **Collar**. Our constructor (of the class **Dog**) calls the constructor of the class **Collar**, which allocates memory for the object:

5. Next, the constructor will **initialize**

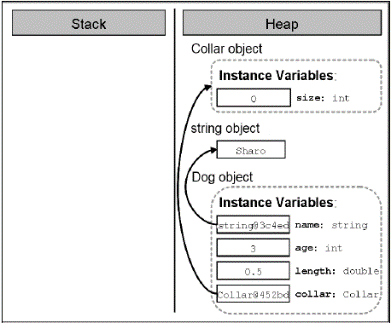
 **it with the default value** for the

respective type. The **size** of the

**Collar** is not explicitly assigned

so it will take the default value for

its type (**0** for **int**):

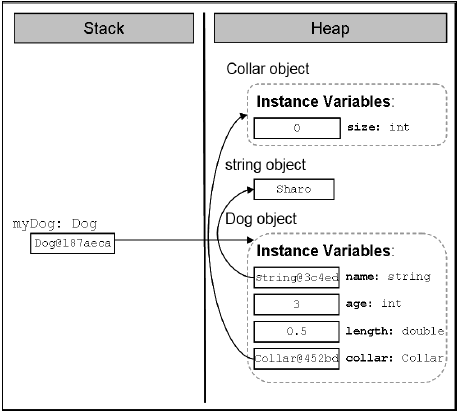
6. After that the reference to the newly

created object, which the constructor of

the class **Collar** returns as a result,

**will be assigned to the field collar**:

7. Finally, the reference to the new object from type **Dog will be assigned to the local variable myDog** in the method **Main()**:

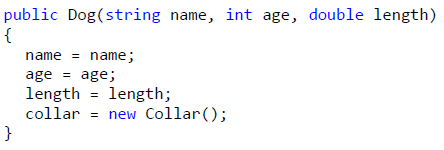


**Info:**

There is no limitation for the number of the constructors of a class in C#. The only requirement is that they **differ in their signature.**

**Scope of Parameters of the Constructor**

**Info:**

Very often, when we declare a constructor with parameters it is possible to name the variables from the parameter list with **the same names** as the names of the fields, which are going to be initialized. Let’s, for example, consider the constructor of the class **Dog**:



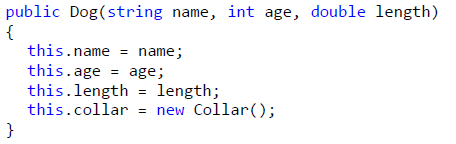
**Info:**

Strange result, isn’t it? In fact this result is not so awkward. The explanation is the following: the scope, in which **the variables from the list of the constructor parameters are acting**, **overlaps** **the scope of acting of the fields** with the same names in the constructor.

Thus, **we do not assign any value to the fields** because in practice we have no access to them. For example,instead of assigning the variable value to the field age, we assign the value of

the variable age to the variable itself.

As we saw from the section "Hiding Fields with Local Variables", to avoid this problem we should access the field, to which we want to assign a value, **using the keyword this**:



**Constructor with Variable Number of Arguments**

**Info:**

1. When we declare a constructor with variable number of arguments, we must use the reserved word **params**, and then insert the type of the parameters, followed by square parentheses.

Finally the name of the array follows, in which array the arguments used for the calling of the method are stored. For example for whole number arguments we can use **params int[] numbers**.

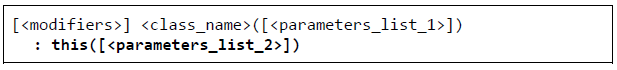
2. It is allowed for the constructor with a variable number of arguments to have other parameters too in the parameter list.

3. The parameter for the variable number of arguments must be the last in the parameter list of the constructor.

**Info:**

Creating **constructors with different signatures** is called **constructor overloading**.

In C# a mechanism exists through which **one constructor can call another** one declared in the same class. This is done again with the keyword **this**, butused in another syntax structure in declaring the constructors:



**When we do not declare any constructor in a given class, the compiler will create one, known as a default implicit constructor!** It will not do anything in addition to the default zeroing of the object fields.

**The default constructor is usually public (**except for some very specific situations, where it is **protected).**

**The default constructor is always without parameters.**

The rule about the default implicit parameterless constructor is:

**If we declare at least one constructor in a given class, the compiler will not create a default constructor for us.**

**Difference between a Default Constructor and a Constructor without Parameters**

**Info:**

Although the default constructor and the one without parameters are similar in signature, they are completely different.

The difference is that the default implicit constructor is created by the compiler, if we do not declare any constructor in our class, and the **constructor without parameters** is declared by us.

Moreover, as explained earlier, **the default constructor will always have** access level **protected** or **public**, depending on the access modifier of the class, while the level of access of the constructor without parameters all depends on us – we define it.

**Properties in C# – Introduction by Example**

**Info:**

Using the properties is a good and proven practice and an important part of the concepts for object-oriented programming.

The creation of a property in programming is done by **declaring two methods** – one for access (**reading**) and one for modifying (**setting**) the value of the respective property.

**Properties – Encapsulation of Fields**

**Info:**

The main objective of the properties is to ensure the **encapsulation of the state of the class** in which they are declared, i.e. to protect the class fromfalling into **invalid state**.

**Encapsulation** is **hiding of the physical representation** of data in one class so that if we subsequently change this presentation, it will not reflect on other classes, which use this class.

**Properties allow further control over the data** in the class and they can check whether the assigned values are correct according to some criteria.

Since the properties are accessed by special methods (called methods for access and modification or **accessor methods**) to be discussed later, for the classes that will use our class the question how the information will be stored would not matter (because of the good encapsulation).

In the most common case, however, the information about the properties of the class is saved in a field of the class, which has the most rigorous level of visibility – **private**.

It does not matter how the information for the properties in a class in C# is saved, **but usually this is done by a class field with the most restrictive access level (private).**

As we will see later, a **property does not necessarily have an accessing and a modifying method at the same time**.

**Method for Reading the Value of a Property (Getter)**

**Info:**

As we explained, the declaration of a **method for reading a value of a property** (in the literature called a **getter**) is made in the body of a propertyby using the following syntaxes:

**get { <accessor\_body> }**

The method of reading the value of a property must end with a **return** or **throw** operation. The type of the value, which is returned as a result of this method, has to be the same as **<property\_type>** described in the property declaration.

**Method for Modifying Property’s Value (Setter)**

**Info:**

Like the method of reading the property’s value we can also declare the method of changing (**modifying) the value of a property** (in the literature known as **setter**). It is declared in the body of a property with **void** return value and the assigned value is accessible through an implicit parameter **value**.

**set { <accessor\_body> }**

**Assertion of the Input Values**

**Info:**

It is a good practice in the programming process to **check the validity of the input values** for the **setter method** of modifying a property and if they are not valid to take the necessary “measures”. Mostly, in case of incorrect input data an exception is caused.

To protect itself from invalid data a class must **verify the input values for all properties and constructors** submitted to the setter methods, as well asall methods, which can change a field of a class.

This programming practice to protect classes from invalid data and invalid internal states is widely used and is a part of the "Defensive Programming" concept.

**Use automatic properties for simple classes** where you want to write less code but have in mind that when you use automatic properties your control over the assigned values is limited. You might have difficulties to add checks for invalid data.

**Types of Properties**

**Info:**

Depending on their definition we can classify the properties as follows:

* **Read-only**, i.e. these properties have only a **get** method as shown by the area of the rectangle.
* **Write-only**, i.e. these properties have only a **set** method, but no method for reading the value of the property.
* And the most common case is **read-write**, where the property has **methods both for reading and for changing the value**.

Some properties are designed to be **read-only**. Others are supposed to support **both read and write operations**. The developers should decide whether someone should be able to change the value of given property and define it as read-only or read / write.

**Write-only** properties are used very rarely.

**Static Classes and Static Members**

**Info:**

We call an element static when it is declared with the modifier **static**. In C# we can declare fields, methods, properties, constructors and classes as static.

In practice **the behavior of the method does not depend of the object state** (the values in the object field).

**What Is a Static Member?**

**Info:**

Formally speaking, a **static member** of the class is every field, property, method or other member, which has a **static** modifier in its declaration. That means that fields, methods and properties, marked as static, belong to the particular class rather than to any particular object of the given class.

Therefore, when we mark a field, method or property as **static**, we can use them without creating any object of the given class.

**Static elements of the class can be used without creating an object of the given class!**

Sometimes, however, we want to have **common fields** for all objects of a given class. To achieve that, we have to use the **static** modifier in the field declarations. As we already said, such fields are called **static fields**. In the literature they are also called **class variables**.

We say that the static fields are **class associated**, rather than associated with any object from the particular class. That means that all objects, created by the description of a class **share** the static fields of the class.

**All objects, created by the description of a given class (that is, instances of a given class), share the static fields of the class!**

The static fields are created when we **try to access them for the first time** (read / modify). After their creation they are initialized with their **default values of their types.**

If during the static field declaration we set an initialization value, it will be assigned to the particular static field. The **initialization executes only once** when the field is accessed for the first time right after the assignment has finished. The next time when the field is accessed that field initialization will not execute.

When we access some static field, an amount of memory will be reserved for it and it will be initialized with its default values.

If the field has initialization during declaration (like it is in our case with the **dogCount** field) this initialization will execute. If we try later to access the field from other part of our program this process will not repeat, because the static field already exists and is initialized.

As we said before, the static variables are **shared between all objects** of the class and do not belong to any object of the particular class. That way any object can access and modify the static field values and in the same time other objects can “see” the modified values.

**Constants**

**Info:**

Like the constants of mathematics, in C# special fields of a class called **constants** can be created. Once declared and initialized **constants always have the same value** for all objects of a particular type.

In C# constants are of two types:

1. Constants the values of which are extracted during the compilation of the program (**compile-time constants**).

2. Constants the values of which are extracted during the execution of the program (**run-time constants**).

Constants, which are declared with special word **const**, are static fields. Nevertheless, the use of modifier **static** is not required (nor allowed by the compiler) in their declaration.

**Although the constants declared with a modifier const are static fields, they must not and cannot use the static modifier in their declaration!**

Let’s pay attention, again: **Constants declared with modifier const must be initialized at the moment of their declaration!**

**A const field of a reference type other than string can only be initialized with null!**

In C#, constants, declared with the modifier **const**, can be only of the following types:

1. Primitive types: **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **char**, **float**, **double**, **decimal**, **bool**.

2. **Enumerated types**

3. **Reference types** (mostly the type **string**).

As we can guess, the only reference type, which can be calculated at compile time while using the operator **new** is **string**.

Therefore, the only possibilities for reference type constants that are declared with modifier const are, as follows:

1. The constants must be of type **string.**

2. The value, which we assign to the constant of reference type**,** other than **string,** is **null.**

**Constants declared with modifier const must be of primitive, enumeration or reference type, and if they are of reference type, this type must be either a string or the value, that we assign to the constant, must be null!**

**Runtime Constants (readonly)**

**Info:**

When we want to declare reference type constants, which cannot be calculated during compilation of the program, we must use a combination of **static readonly** modifiers, instead of **const** modifier.

The examples made it clear that the difference between **const** and **static readonly** fields is in the moment of their value assignments. The compiletime constants (**const**) must be initialized at the moment of declaration, while the run-time constants (**static readonly**) can be initialized at a later stage, for example in one of the constructors of the class in which they are defined.

Constants are used in programming to **avoid repetition of numbers, strings or other common values** (literals) in the program and to enablethem to change easily.

According to some authors all literals other than **0**, **1**, **-1**, empty string, **true**, **false** and **null** must be declared as constants, but this can make it difficult to read and maintain the code instead

of making it simple.

Therefore, it is believed that **values, which occur more**

**than once in the program or are likely to change over time, must be declared as constants**.

**Static Methods**

**Info:**

Like static fields, we declare a method as static if we want it to be associated only with the class and not with a particular class object.

In most cases **static methods are used to access static fields** in the class they have been defined.

We can access static fields and static methods of the class from non-static method. As we learned earlier, this is because static methods and variables are bound by class, rather than a specific method and the static elements can be accessed from any object of the class, even of external classes (as long as they are visible to them).

We can call a static method or static field of the class from another static method without any problems.

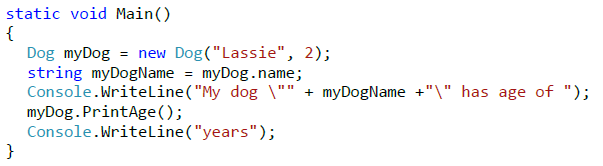
We should know that from static method we can neither access non-static fields, nor call non-static methods. This is because static methods are bound to the class and do not “know” any object of the class.

**Therefore, the keyword** this **cannot be used in static methods – it is bound to a specific instance of the class!**

When we try to access non-static elements of the class (fields or methods) from static method, we will always get a compilation error.

**Non-static elements of the class may NOT be used in a static context!**

The problem with the access to non-static elements of the class of static method has a single solution – these non-static elements are accessed by reference to an object:



**The keyword this cannot be used in static properties!**

**Static properties can be accessed only through dot notation, applied to the name of the class in which they are declared!**

When a class is declared as static, it is an indication that **this class contains only static members** (i.e. static fields, methods, properties) and **cannot be instantiated.**

The use of static classes is rare and most often associated with the **use of static methods and constants**, which do not belong to any particularobject.

**Static Constructors**

**Info:**

Static constructors can be declared both in static and in non-static classes. They are **executed only once** when the first of the following two events occurs for the first time:

1. **An object of class is created.**

2. **A static element of the class is accessed** (field, method, property).

Most often static constructors are used for initialization of static fields.

**Structures**

Info:

In C# and .NET Framework there are two implementations of the concept of "class" from the object-oriented programming: **classes** and **structures**.

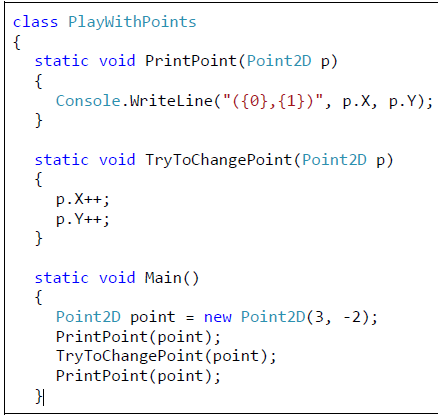
* The main difference between a structure and a class is that: **Classes are reference types** (references to some address in the heap which holds their members).
* **Structures (structs) are value types** (they directly hold their members in the program execution stack).

They are **value types** (**not objects**), which means they cannot be **null** and they are **passed by value** when taken as a method parameters.

**Structures are Value Types**

**Info:**

Unlike classes, the **structures are value types**. To illustrate this we will play a bit with the **Point2D** structure:



**struct** Point2D

{

private double x;

private double y;

public Point2D(int x, int y)

{

this.x = x;

this.y = y;

}

public double X

{

get { return this.x; }

set { this.x = value; }

}

public double Y

{

get { return this.y; }

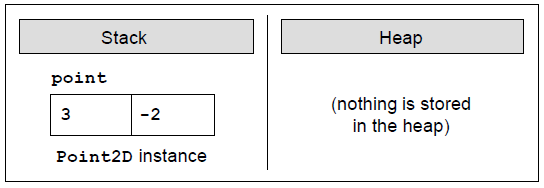
set { this.y = value; }

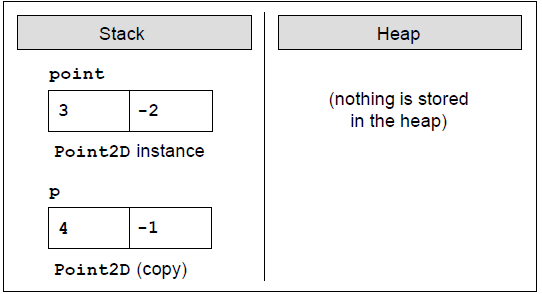
}

}

Obviously the **structures are value types** and when passed as parameters to a method **their fields are copied** (just like **int** parameters) and when changed inside the method, the change affects only the copy, not the original. This can be illustrated by the next few figures.

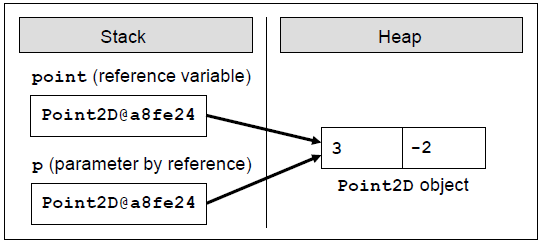
1. First, the **point** variable is created which holds a value of (3, -2):



2. Next, the method **TryToChangePoint(Point2D p)** is called and it copies the value of the variable **point** into **another place in the stack**, allocated for the parameter **p** of the method. When the parameter **p** is changed in the method’s body, it is modified in the stack and this **does not affect the original variable point** which was previously passed as argument when calling the method:

3. If we change **Point2D** from **struct** to **class**, the result will be very different:

This is because the variable **point** will be now passed by reference (not by value) and its value will be shared between **point** and **p** in the heap. The figure below illustrates what happens in the memory at the end of the method **TryToChangePoint(Point2D p)** when **Point2D** is a class:



Classes are used more often than structures. Use structs as exception, and **only if you know well what are you doing**!

**Enumerations**

**Info:**

**Enumeration** is a **structure**, which resembles a class but differs from it in that **in the class body we can declare only constants.**

Enumerations can take values only from the constants listed in the type. An enumerated variable can have as a value one of the listed in the type constants but cannot have value **null**.

Constants separated by commas are declared in the enumeration block.

**The enumerations are a set of constants of type – this listed type!**

**Nature of Enumerations**

**Info:**

Each **constant**, which is declared in one enumeration, **is being associated with a certain integer. By default**, for this **hidden integer representation of constants** in one enumeration **int is being used**.

To show “the integer nature” of constants in the listed types let’s try to figure out what’s the numerical representation of the constant, which corresponds to “Monday” from the example of the previous subsection:

The values, associated with constants of a particular enumerated type by default are the indices in the list of constants of this type, i.e. numbers from 0 to the number of constants in the type less 1.

**Each constant in one enumeration is actually a textual representation of an integer. By default this number is the constant’s index in the list of constants of a particular enumeration type!**

**Use of Enumerations**

**Info:**

The main purpose of the enumerations is to **replace the numeric values**, which we would use, if there were no enumeration types. In this way the code ecomes simpler and easier to read.

Another very important application of the enumerations is the pressure exercised by the compiler to use constants from the enumerations and not just numbers.

**The constants of enumerations can be used in switch-case structures!**

**Whenever possible, use enumerations instead of set of constants declared in a class!**

**When we modify the list of constants in an existing enumeration, we should be careful not to break the logic of the code that already exists and uses the constants, declared so far!**

**Inner Classes (Nested Classes)**

**Info:**

In C# an inner (nested) class is called a **class that is declared inside the body of another class**. Accordingly, the class that encloses the inner class is called an **outer class**.

The main reason to declare one class into another are:

1. To **better organize the code** when working with objects in the real world, among which have a special relationship and one cannot exist without the other.

2. To **hide a class in another class**, so that the inner class cannot be used outside the class wrapped it.

In **general**, **inner classes are used rarely**, because **they complicate the structure of the code** and **increase the nested levels.**

Allowed modifiers in the declaration of the class are:

1. **public** – an inner class is accessible from any **assembly**.

2. **internal** – an inner class is available in the **current** **assembly**, **in which is located the outer class**.

3. **private** – access is restricted only to the **class holding the inner class.**

4. **static** – an **inner class contains only static members.**

There are four more permitted modifiers – **abstract, protected, protected internal**, **sealed** and **unsafe**, which are outside the scope and subject of this chapter and will not be considered here.

The keyword **this** to an inner class has relation only to the internal class, but not to the outside. Fields of the outside class **cannot be accessed** using the reference **this**. If necessary fields of the outer class can be accessed by the internal, it needs in creating the internal class to submit a reference to an outer class.

**Static members** (fields, methods, properties) of the outer class **are accessible from the inner class** regardless of their level of access!

**When the connection between the two classes is a composition, the class, which consequently is a part of another class, is convenient to be declared as inner class!**

Before proceeding to the next section that refers to generic types, it should be noticed, that sometimes **enumeration should and can be declared within** **a class** in order of better encapsulation of the class!

**Generics**

**Info:**

**What Is a Generic Class?**

Like the methods, when we know, that the functionality (actions) encapsulated into a class, can be applied not only to objects of one, but to many (heterogeneous) types, and these types are not known at the time of declaring the class, we can use a functionality of the language C# called **generics** (generic types).

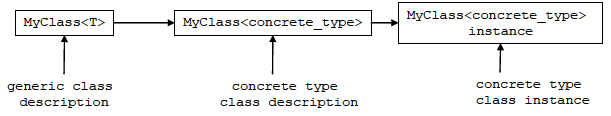
It allows us to **declare parameters of this class, by indicating an unknown type** that the class will work eventually with.

**Typifying a class (creating a generic class) means to add to the declaration of a class a parameter (replacement) of unknown type, which the class will use during its operation. Subsequently, when the class is instantiated, this parameter is replaced with the name of some specific type!**

**Typifying (Generics) – Behind the Scenes**

**Info:**

Before we continue, let’s us explain what happens into the memory of the computer, when we work with generic classes.



**First** we declare our generic class **MyClass<T>** (generic class description in the scheme above).

**Then** the compiler translates our code to an intermediate language (MSIL), as translated code contains information that the class is generic, i.e. it works with undefined types until now.

**At runtime**, when someone tries to work with our generic class and tries to use it with a specific type, a new **description of the class** is created (specific type class description in the diagram above), which is identical to the generic class, with the difference that where it has been used **T**, now is replaced by a specific type.

**For example**, if you try to use **MyClass<int>**, everywhere in your code, where the unknown parameter **T** is used, it will be replaced with **int**.

**Only then** we can create object of a generic class with a specific type **int**.

**The interesting thing** here is that to create this object, the description of the class, which was created in the meantime (specific type class description), will be used.

**Instantiating of a generic class** by given specific types of its parameters is called "**specialization of the type**" or "**extension of generic class**".

**Generic Methods**

**Info:**

Like classes, when the type of method’s parameters cannot be specified, we can **parameterize (typify) the method**. Accordingly,

**Typifying of a method** is done, when after the name and before the opening bracket of the method, we add **<K>**, where **K** is the replacement of the type that will be used later:



\

For example, consider a **method that swaps the values of two variables**:

public void Swap<K>(ref K a, ref K b)

{

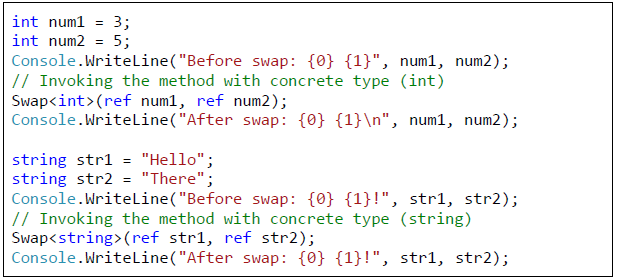
K oldA = a;

a = b;

b = oldA;

}

This is a method that swaps the values of two variables, **without carrying of their types**. That is why we define it as a generic, so we can use it for alltypes of variables.



**Info:**

We notice that in the list of parameters we have used also the keyword **ref**. This concerns the specification of the method – namely, to exchange the values of two references.

By using the keyword **ref**, the method will use the

same reference that was given by the calling method. This way, all changes on this variable made by our method, will remain after the method exits.

We should know that by **calling a generic method**, we can miss the explicit declaration of a specific type (in our example **<int>**), because the compiler will detect it automatically, recognizing the type of the given parameters.

In other words, our code can be simplified using the following calls:



It should be noticed that static methods can also be typified, unlike the properties and constructors of the class.

**Static methods can also be typified, but properties and constructors of the class cannot!**

**Using a Keyword "default" in a Generic Source Code**

**Info:**

To handle this problem, in our code we have to use the construct **default(T)** instead of **null**, which returns the default value for the particular type that will be used instead of T.

As we know, the default value for reference type is **null**, and for numeric types – zero.

Generic classes and methods **increase the reusability of the code**, the

security and the performance compared to other non-generic alternatives.

As a general rule, the **programmer should strive to create and use generic classes, whenever it is possible!**

**The more generic types are used, the higher level of abstraction there is in the program and the source code becomes more flexible and reusable!**

We should keep in mind, that overuse of generics can lead to over-generalization and the code may become unreadable and difficult to understand by other programmers.

**Text Files**

**Info:**

What is a **stream**?

A **stream** is an **ordered sequence of bytes**, which is send from one application or input device to another application or output device.

Streams are an **abstraction of a data communication channel that connects two devices or applications**.

Every time when you read or write from or to a file, you have to **open a stream** to the corresponding file, **do the reading or writing**, and then **close the stream**.

There are two types of streams – **text streams** and **binary streams** but this separation has to do with the interpretation of the sent and received bytes.

Streams and media streaming are different concepts but both use **sequences of data**.

Streams **do not allow random access** to their data, only **sequential**.

Different situations require **different types of streams**. Some streams are used with **text files**, others-with **binary files** and then there are those that work with **strings**.

**Closing the stream is very important** and must not be left out, because you risk losing data, damagingthe file, to which the stream is opened, and so on – all of these are verytroublesome scenarios, which must not happen in our programs.

From one side **we pour data in** and from the other **data leaks out.** Therefore, we can consider streams as a **data transport channel**, such as pipes.

**Streams in .NET – Basic Classes**

**Info:**

In .NET Framework **classes for working with streams** are located in the namespace **System.IO.** We can distinguish two main types of streams – those who work with **binary data** and those who work with **text data.**

At the top of the stream hierarchy stands an **abstract input-output stream class**. It cannot be instantiated but defines the basic functionality that all the other streams have.

There are **buffered streams** that do not add any extra functionality, but use a buffer for reading and writing data, which significantly enhances performance.

Some streams **add additional functionality** to reading and writing data. For example, there are streams that compress / decompress data sent to them and streams that encrypt / decrypt data.

The main classes in the **System.IO** namespace are **Stream** (abstract base class for all streams in .NET Framework), **BufferedStream**, **FileStream**, **MemoryStream**, **GZipStream** and **NetworkStream**.

All streams in C# are similar in one basic thing – **it is mandatory to close them** after we have finished working with them.

**Always close the streams and files you work with! Leaving an open stream or file leads to loss of resources and can block the work of other users or processes in your system!**

**Avoid full file paths and work with relative paths! This makes your application portable and easy for installation and maintenance!**

**Remember that when you start the C# program, the current directory is the one, in which the executable (.exe) file is located. Most often this is the subdirectory bin\Debug or bin\Release directory to the root of the project. Therefore, to open the file example.txt from the root directory of your Visual Studio project, you should use a relative path ..\..\example.txt.**

**Universal Relative to Physical Path Resolver**

**Info:**

It can automatically **resolve a relative path to full (physical) file path** in Web, desktop, console or other .NET application.

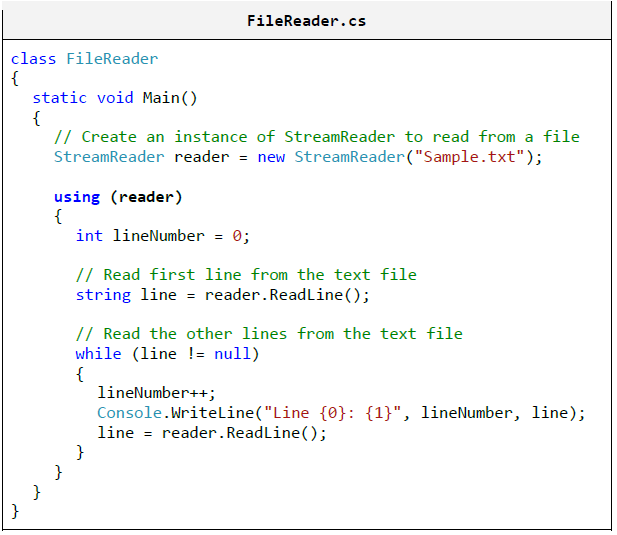
string logoPath = UniversalFilePathResolver.ResolvePath(@"~\logo.gif");

**Always close the StreamReader instances after you finish working with them. Otherwise you risk losing system resources. Use the method Close() or the statement using!**

**Automatic Closing of the Stream after Working with It**

**Info:**

The C# construct **using(…)** ensures that after leaving its body, the method **Close() will automatically be called**. This will happen even if an exception occurs when reading the file.



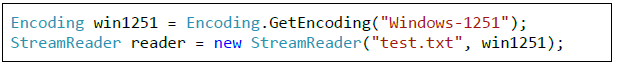
**Always use the using construct**

**in C# in order to properly close**

**files and streams!**

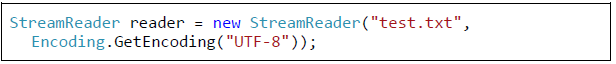
For working with encodings (charsets) in .NET Framework, the class **System.Text.Encoding** is used, which is created the following way:





If you do not explicitly set the encoding scheme (encoding) for the file read, in .NET Framework, the default encoding **UTF-8** will be used.

**To avoid problems with incorrect encoding of files, always check the encoding explicitly. Otherwise, you may work incorrectly or break at a later stage!**



**Writing to a Text File**

**Info:**

**Be sure to close the stream after you finish using it! The best way to dispose any unused resources is with the using construct in C#.**

**Linear Data Structures**

**Info:**

Some of the basic presentations of data in programming: **lists and linear data structures**. Very often in order to solve a given problem we need to work with a sequence of elements.

**Abstract Data Structures**

**Info:**

* What is a **Data Structure?**

Very often, when we write programs, we have to work with many objects (data). **Data structures** – **a set of data** organized on the basis of logical and mathematical laws.

Very often the choice of the right data structure makes the program much more efficient – we could save memory and execution time.

* What is an **Abstract Data Type?**

**Abstract data types (ADT)** gives us a definition (abstraction) of the specific structure, i.e. defines the allowed operations and properties, without being interested in the specific implementation.

* We can differentiate **several groups of data structures**:

1. **Linear** – these include lists, stacks and queuess

2. **Tree-like** – different types of trees like binary trees, B-trees and balanced trees

3. **Dictionaries** – key-value pairs organized in hash tables

4. **Sets** – unordered bunches of unique elements

5. **Others** – multi-sets, bags, multi-bags, priority queues, graphs, …

**Mastering basic data structures in programming is essential**, as without them we could not program efficiently.

**List Data Structures**

**Info:**

Most commonly used data structures are the **linear (list) data structures.** They are an abstraction of all kinds of rows, sequences, series and others from the real world.

**1. List**

**Info:** We could imagine the **list** as an **ordered sequence (line) of elements.**

**Abstract Data Structure (ADT) "List"**

**Info:**

**List** is a **linear data structure**, which contains a sequence of elements. The list has the property **length** (count of elements) and its elements are **arranged consecutively.**

Like we already mentioned, an **ADT** can have several implementations. An example of such ADT is the **interface System.** **Collections.IList.**

Each **ADT** defines some **interface**. Let’s consider the interface **System.Collections.IList**. The basic methods, which it defines, are:

* **int Add(object)** – adds element in the end of the list;
* **void Insert(int, object)** – adds element on a preliminary chosen position in the list
* **void Clear()** – removes all elements in the list
* **bool Contains(object)** – checks whether the list contains the element
* **void Remove(object)** – removes the element from the list
* **void RemoveAt(int)** – removes the element on a given position
* **int IndexOf(object)** – returns the position of the element
* **this[int]** – indexer, allows access to the elements on a set position

**Static List (Array-Based Implementation)**

**Arrays**

**Info:**

**Arrays** perform many of the features of the ADT list, but there is a significant difference – the lists allow adding new elements, while arrays have fixed size.

Despite of that, an implementation of list is possible with an array, which automatically increments its size (similar to the class **StringBuilder**. Such list is called **static list** **implemented with an extensible array**.

**Linked List (Dynamic Implementation)**

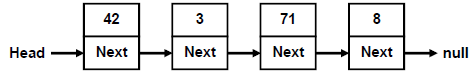
**Info:**

As we saw, the **static list** has a serious disadvantage – the operations for inserting and removing items from the inside of the array requires rearrange-ment of the elements.

When frequently inserting and removing items (especially a large number of items), this can lead to low performance. In such cases it is advisable to use the so called **linked lists**.

The difference in them is the structure of elements – while in the static list the element contains only the specific object, with the dynamic list the **elements keep information about their next element**.

Here is how a **sample linked list** looks like in the memory:



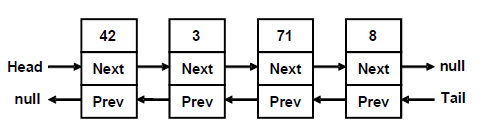
The above example demonstrates that certain ADT could be implemented in several conceptually different ways and the users may not notice the difference between them. Still, different implementations could have different performance and could take different amount of memory.

This concept, known as **abstract behavior**, is fundamental for OOP and can be implemented by **abstract classes** or **interfaces** as we shall see in the section "Abstraction" of chapter "Object-Oriented Programming Principles".

**Doubly-Linked List**

**Info:**

In the so called **doubly-linked lists** each element contains its **value** and **two pointers – to the previous and to the next element** (or **null**, if there is no such element).



**2. The ArrayList Class**

**Info:**

The first one is the class **ArrayList**, which is an **untyped dynamically-extendable array**. It is implemented similarly to the static list implementation, which we considered earlier. **ArrayList** gives the opportunity to add, delete and search for elements in it.

Some more important class members we may use are:

* **Add(object)** – adding a new element
* **Insert(int, object)** – adding a new element at a specified position (index)
* **Count** – returns the count of elements in the list
* **Remove(object)** – removes a specified element
* **RemoveAt(int)** – removes the element at a specified position
* **Clear()** – removes all elements from the list
* **this[int]** – an indexer, allows accessing the elements by a given position (index)

As we saw, one of the main problems with this implementation is the resizing of the inner array when adding and removing elements.

In the **ArrayList** the problem is solved by preliminarily created array (buffer), which gives us the opportunity to add elements without resizing the array at each insertion or removal of elements.

The **ArrayList** class is **untyped**, so it **can keep all kinds of elements** – numbers, strings and other objects!

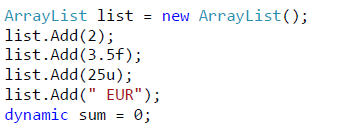
**ArrayList of Numbers – Example**

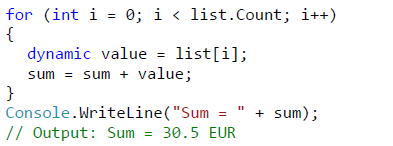
**Info:**

In C# **dynamic** is a universal data type intended to hold any value (numbers, objects, strings, even functions and methods).

Operations over **dynamic** variables (like the **+** operator used above) are **resolved at runtime** and their action depends on the actuals values of their arguments. At compile time almost every operation with **dynamic** variables successfully compiles.

At runtime, if the operation can be performed, it is performed, otherwise and exception is thrown. This explains why we apply the operation **+** over the arguments **2**, **3.5f**, **25u** and **" EUR"** and we finally obtain as a result the string **"30.5 EUR"**.





**Generic Collections**

**Info:**

To solve the problem we use the **generic (template / parameterized) classes**. They are created to work with one or several types, as when wecreate them, we indicate what type of objects we are going to keep in them.

GenericType<T> instance = new GenericType<T>();

This type **T** can be any successor of the class **System.Object**, for example **string** or **DateTime**.

**3. The List<T> Class**

**Info:**

**List<T> is the generic variant of ArrayList**. When we create an object of type **List<T>**, we indicate the type of the elements, which will be hold in the list, i.e. we substitute the denoted by **T** type with some real data type (for example number or string).

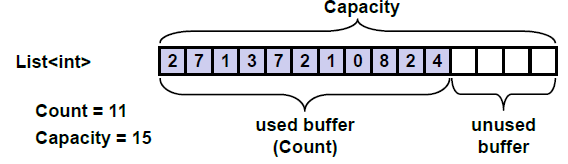
Let’s consider a case in which we would like to create a list of integer elements:

List<int> intList = new List<int>();

**The List Class – Array-Based Implementation**

**Info:**

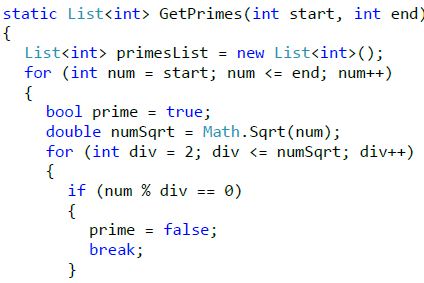
It keeps its elements in the memory as an array, which is **partially in use and partially free** for new elements (blank). We could imagine a **List<T>** like an **array, which has some capacity and is filled to a certain level**:

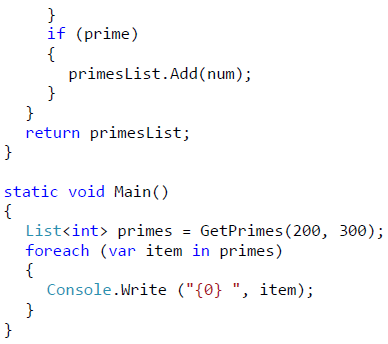


We could say that **List<T>** combines the good sides of lists and arrays – fast adding, changeable size and direct access by index.

**Use List<T> when you don’t expect frequent insertion and deletion of elements, but you expect to add new elements at the end of the list or to access the elements by index!**

**Prime Numbers in Given interval - Example**

After we got familiar with the implementation of the structure list and the class List<T>, let’s see how to use them. We are going to consider the problem for finding the prime numbers in a certain interval. For this purpose we have to use the following algorithm:

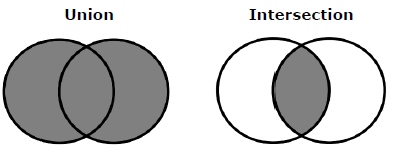


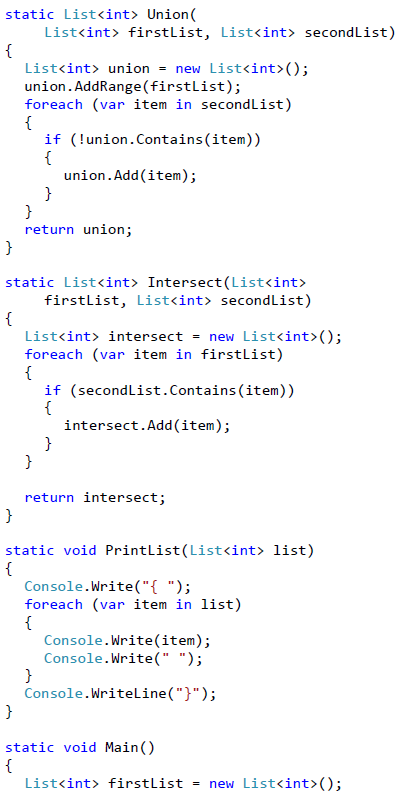
From the mathematics we know that if a number is **not prime** it has **at least one divisor** in the interval [2 … square root from the given number]. This is what we use in the example above. For each number we look for a divisor in this interval. If we find a divisor, then the number is not prime and we could continue with the next number from the interval. Gradually, by adding the prime numbers we fill the list, after which we traverse it and print it on the screen. Here is how the output of the code above looks like:

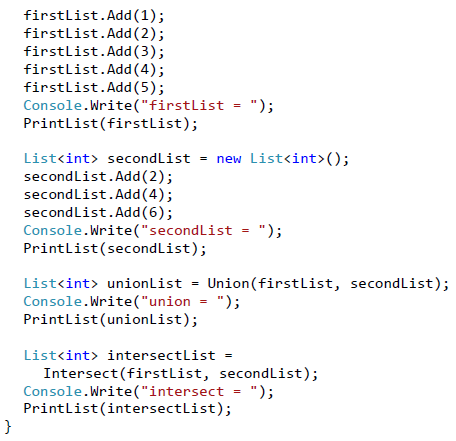


**Union and Intersection of Lists – Example**

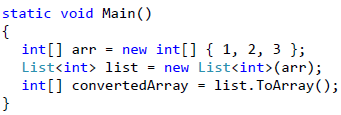
**Info:**

Let’s consider a more interesting example – let’s write a program, which can find the **union** and **the intersection** of **two sets of numbers.**





In C# the **conversion of a list to an array** is easy by using the given method **ToArray**(). For the opposite operation we could use the constructor of **List<T>(System.Array).**



**4. The LinkedList<T> Class**

**Info:**

This class is a **dynamic implementation of a doubly linked list** built in .NET Framework. Its elements contain a certain value and a pointer to the previous and the next element. The **LinkedList<T>** class in .NET works in similar fashion like our class **DynamicList<T>**.

**Basic Operations in the LinkedList<T> Class**

**Info:**

**LinkedList<T>** has the same operations as in **List<T>**, which makes the two classes interchangeable, but in fact **List<T>** is used more often. Later we are going to see that **LinkedList<T>** is used when working with queues.

1. **Stack**

**Info:**

Let’s imagine several cubes, which we have put one above other. We could put a new cube on the top, as well as remove the highest cube. Or let’s imagine a chest. In order to take out the clothes on the bottom, first we have to take out the clothes above them.

In programming the **stack is a commonly used data structure**.

The stack is used internally by the .NET virtual machine (CLR) for keeping the variables of the program and the parameters of the called methods (it is called **program execution stack**).

The **stack** is a data structure, which implements the behavior "**last in – first out**" (**LIFO**). As we saw with the cubes, the elements could be added andremoved only on the top of the stack.

**ADT stack provides 3 major operations:**

* **push** (add an element at the top of the stack);
* **pop** (take the last added element from the top of the stack);
* **peek** (get the element form the top of the stack without removing it).

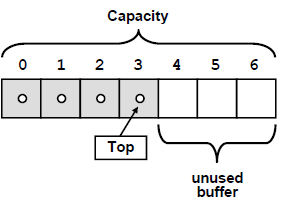
The data structure stack can also have different implementations, but we are going to consider two – **dynamic** and **static** **implementation**.

**Static Stack (Array-Based Implementation)**

**Info:**

Like with the static list we can use an array to keep the elements of the stack. We can keep an index or a pointer to the element, which is at the top.

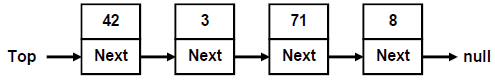
Here is how we could imagine a **static stack**:



**Linked Stack (Dynamic Implementation)**

**Info:**

For the **dynamic implementation of stack** we use elements, which keep a value and a pointer to the next element. This **linked-list based implementation** does not require an internal buffer, does not need to grow when the buffer is full and has virtually the same performance for the major operations like the static implementation:



When the stack is empty, the **top** has value **null**. When a new item is added, it is inserted on a position where the **top** indicates, after which the top is redirected to the new element. Removal is done by deleting the first element, pointed by **the** top pointer.

1. **The Stack<T> Class**

**Info:**

In C# we could use the standard implementation of the class in .NET Framework:

* **System.Collections.Generics.Stack<T>**

It is implemented statically with an **array**, as **the array is resized when needed.**

**The Stack<T> Class – Basic Operations**

**Info:**

All basic operations for working with a stack are implemented:

* **Push(T)** – adds a new element on the top of the stack
* **Pop()** – returns the highest element and removes it from the stack
* **Peek()** – returns the highest element without removing it
* **Count** – returns the count of elements in the stack
* **Clear()** – retrieves all elements from the stack
* **Contains(T)** – check whether the stack contains the element
* **ToArray()** – returns an array, containing all elements of the stack

1. **Queue**

**Info:**

The "**queue**" data structure is **created to model queues**, for example a queue of waiting for printing documents, waiting processes to access a common resource, and others.

The **abstract data structure "queue"** satisfies the behavior "**first in – first out**" (**FIFO**)!

Like with the lists, the **ADT queue** could be implemented **statically** (as resizable array) and **dynamically** (as pointer-based linked list).

**Static Queue (Array-Based Implementation)**

**Info:**

In the **static queue** we could use an **array** for keeping the elements. When adding an element, it is inserted at the index, which follows the end of queue.

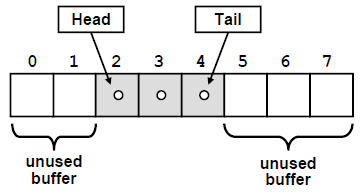
* After that **the end points at the newly added element.**

When removing an element, we take the element, which is pointed by the head of the queue.

* After that **the head starts to point at the next element.**

Thus **the queue moves to the end of the array**. When it reaches the end of the array, when adding a new element, it is inserted at the beginning of the array.

That is why the implementation is called "**looped queue**", as we mentally stick the beginning and the end of the array and the queue orbits it:



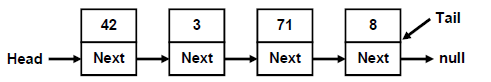
Static queue keeps an **internal buffer** with bigger capacity than the actual number of elements in the queue. Like in the static list implementation, when the space allocated for the queue elements is finished, the **internal buffer grows** (usually doubles its size).

The **major operations** in the queue ADT are **enqueue** (append at the end of the queue) and **dequeue** (retrieve an element from the start of the queue).

**Linked Queue (Dynamic Implementation)**

**Info:**

The **dynamic implementation** of queue ADT looks like **the implementation of the linked list**. Like in the linked list, the elements consist of two parts – **a value and a pointer to the next element**:



However, here elements are **added at the end** of the queue (**tail**), and are **retrieved from its beginning** (**head**), while we have no permission to get or add elements at any another position.

1. **The Queue<T> Class**

**Info:**

In C# we use the **static implementation** of queue via the **Queue<T> class**. Here we could indicate the type of the elements we are going to work with, as the queue and the linked list are generic types.

**The Queue<T> – Basic Operations**

**Info:**

**Queue<T>** class provides the basic operations, specific for the data structure queue. Here are some of the most frequently used:

* **Enqueue(T)** – inserts an element at the end of the queue
* **Dequeue()** – retrieves the element from the beginning of the queue and removes it
* **Peek()** – returns the element from the beginning of the queue without removing it
* **Clear()** – removes all elements from the queue
* **Contains(T)** – checks if the queue contains the element
* **Count** – returns the amount of elements in the queue

The queue is **FIFO structure (first-in, first out)**!

**Trees and Graphs**

**Info:**

Each of this data structures is used for building a model of **real life problems**, which are efficiently solved using this model.

The focus it will be on **binary trees**, **binary search trees** and **self-balancing binary search tree**.

**1. Tree Data Structures**

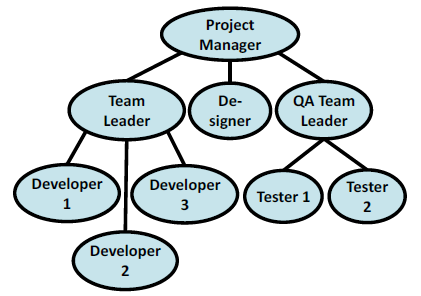
**Info:**

A **tree-like data structure** or **branched data structure** consists of set of **elements** (**nodes**) which could be linked to other elements, sometimes **hierarchically**, sometimes **not**.

**Trees** represent **hierarchies**, while **graphs** represent **more general relations** such as the map of city.

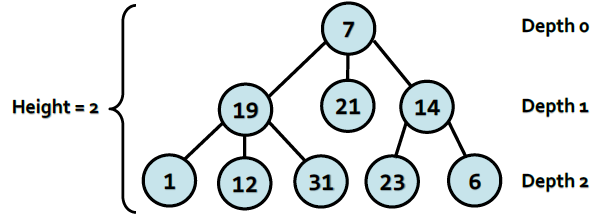
* 1. **Trees**

**Info:**

 **Trees** are very often used in programming, because they naturally represent all kind of **object hierarchies** from our surroundings.

**Trees Terminology**

**Info:**



We will call every circle a **node** and each line an **edge**. Nodes "19", "21", "14" are below node "7" and are directly connected to it. This nodes we are called **direct descendants (child nodes)** of node "7", and node "7" their **parent**.

Intuitively we can say that "21" is **sibling** of "19", because they are both children of "7" (the reverse is also true – "19" is sibling of "21").

For "1", "12", "31", "23" and "6" node "7" precedes them in the hierarchy, so he is their indirect parent – **ancestor**, ant they are called his **descendants**.

**Root** is called the **node without parent**. In our example this is node "7".

**Leaf** is a **node without child nodes**. In our example – "1", "12", "31", "21", "23" and "6".

**Internal nodes** are the nodes, which are **not leaf or root** (all nodes, which have parent and at least one child). Such nodes are "19" and "14".

**Path** is called a **sequence of nodes connected with edges**, in which there is no repetition of nodes. Example of path is the sequence "1", "19", "7" and "21".

The sequence "1", "19" and "23" is not a path, because "19" and "23" are not connected.

**Path length** is the number of edges, connecting the sequence of nodes in the path.

Actually it is equal to the **number of nodes in the path minus 1**. The length of our example for path ("1", "19", "7" and "21") is three.

**Depth** of a node we will call the **length of the path from the root to certain node**.

In our example "7" as **root** has depth **zero**, "19" has depth **one** and "23" – depth **two**.

**Def1:**

**Tree** – a **recursive data structure**, which consists of **nodes, connected with edges**. The following statements are true for trees:

* Each node can have **0 or more direct descendants (children).**
* Each node has **at most one parent**. There is only one special node without parent – **the root** (if the tree is not empty).
* All nodes are **reachable from the root** – there is a path from the root to each node in the tree.

**Def2:**

We can give more simple definition of tree: **a node is a tree and this node can have zero or more children, which are also trees**.

**Height** of tree – is the **maximum depth of all its nodes**. In our example the tree height is 2.

**Degree** of node we call the **number of direct children** of the given node.

The degree of "19" and "7" is three, but the degree of "14" is two. The leaves have degree zero.

**Branching factor** is the **maximum of the degrees of all nodes** in the tree.

In our example the maximum degree of the nodes is 3, so the branching factor is 3.

**Creating a Tree**

**Info:**

We to make **creating** **a tree** easier we defined a **special constructor**, which takes for input parameters **a node value and a list of its sub-trees**.

That allows us to give any number of arguments of type **Tree<T>** (sub-trees). We used exactly the same constructor for creating the example tree.

**Tree Implementation – Example**

**Info:**

**Depth-First-Search (DFS) Traversal**

**Info:**

The **Depth-First-Search algorithm** aims to visit each of the tree nodes exactly one. Such a visit of all nodes is called **tree traversal**. There are multiple algorithms to traverse a tree but in this chapter we will discuss only two of them: **DFS** (depth-first search) and **BFS** (breadth-first search).

The **DFS algorithm** starts from a given node and goes as deep in the tree hierarchy as it can. When it reaches a node, which has no children to visit or all have been visited, it returns to the previous node.

We can describe the depth-first search algorithm by the following simple steps:

1. **Traverse the current node** (e.g. print it on the console or process it in some way).

2. Sequentially **traverse recursively each of the current nodes’ child nodes** (traverse the sub-trees of the current node). This can be done bya recursive call to the same method for each child node.

**Breath-First-Search (BFS)**

**Info:**

Let’s have a look at another way of traversing trees. **Breath-First-Search (BFS)** is an algorithm for traversing branched data structures (like trees and

graphs).

The BFS algorithm first traverses the start node, then all its direct children, then their direct children and so on.

This approach is also known as the **wavefront traversal**, because it looks like the waves caused by a stone thrown into a lake.

The **Breath-First-Search (BFS) algorithm** consists of the following steps:

1. Enqueue the start node in queue **Q**.

2. While Q is not empty repeat the following two steps:

* Dequeue the next node v from Q and print it.
* Add all children of v in the queue.

**The BFS algorithm is very simple** and always traverses first the nodes that are closest to the start node, and then the more distant and so on until it reaches the furthest.

The BFS algorithm is very widely used in problem solving, e.g. for **finding the shortest path in a labyrinth**.

**Binary Trees**

**Info:**

The **binary tree** type turns out to be very useful in programming. The terminology for trees is also valid about binary trees.

**Def1:**

**Binary Tree** – a tree, which nodes have a **degree equal or less than 2** or we can say that it is a tree with **branching degree of 2**.

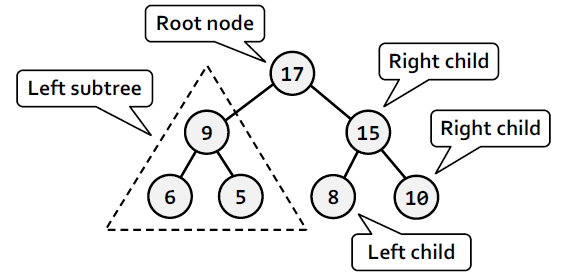
Because every node’s children are at most 2, we call them **left child** and **right child**.

They are the roots of the **left sub-tree** and the **right sub-tree** of their parent node.

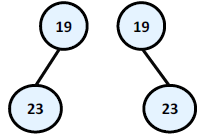
Some nodes may have only left or only right child, not both. Some nodes may have no children and are called **leaves**.

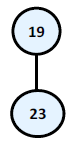
**Def2:**

Binary tree can be **recursively** defined as follows: **a single node is a binary tree and can have left and right children which are also binary trees**.



**Info:**

We have to note that there is one very big difference in the definition of binary tree from the definition of the classical tree – the **order of the children of each node**. The next example will illustrate that difference:

On this figure above two totally different **binary trees** are illustrated – the first one has **root** "19" and its **left child** "23" and the second **root** "19" and **right child** "23".

If that was an **ordinary tree** they

would have been the same. That’s why such

**tree** we would illustrate the following way:

**Remember! Although we take binary trees as a special case of a tree structure, we have to notice that the condition for particular order of children nodes makes them a completely different structure!**

**Binary Tree Traversal**

**Info:**

The **traversal of binary tree** is a classic problem which has classical solutions. Generally there are few ways to traverse a binary tree recursively:

**1. In-order (Left-Root-Right)** – the traversal algorithm first traverses the left sub-tree, then the root and last the left sub-tree. In our example the sequence of such traversal is: "23", "19", "10", "6", "21", "14", "3", "15".

**2. Pre-order (Root-Left-Right)** – in this case the algorithm first traverses the root, then the left sub-tree and last the right sub-tree. The result of such traversal in our example is: "14", "19", "23", "6", "10", "21", "15", "3".

**3.** **Post-order (Left-Right-Root)** – here we first traverse the left subtree, then the right one and last the root. The result after the traversal is: "23", "10", "21", "6", "19", "3", "15", "14".

**Ordered Binary Search Trees**

**Info:**

As examples for a useful properties we can give the ability to quickly search of an element by given value (**Red-Black tree**); order of the elements in the tree (**ordered search trees**); balanced depth (**balanced trees**); possibility to store an ordered tree in a persistent storage so that searching of an element to be fast with as little as possible read operations (**B-tree**), etc.

A more specific class of binary trees are: – **ordered trees**. They use one often met property of the nodes in the binary trees – **unique identification key** in every node.

Important property of these keys is that they are comparable. Important kind of ordered trees are the so called "**balanced search trees**".

**Def:**

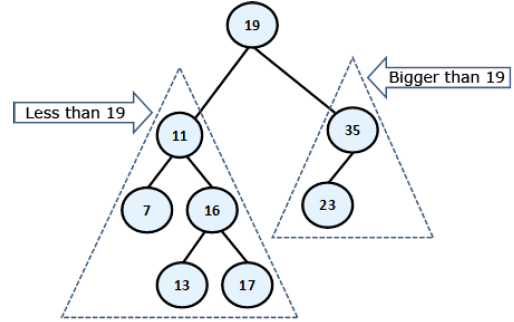
**Ordered Binary Tree** (**binary search tree**) is a **binary tree**, in which **every node has a unique key**, **every two of the keys are comparable** and the tree is organized in a way that for every node the following is satisfied:

1. All keys in **the left sub-tree** are **smaller** than its key.

2. All keys in **the right sub-tree** are **bigger** than its key.

**Properties of the Ordered Binary Search Trees**

**Info:**

On the figure below we have given an **example of an ordered binary search tree**. We will use this example, to give some important properties of the binary tree’s order:

**Comparability between Objects in C#**

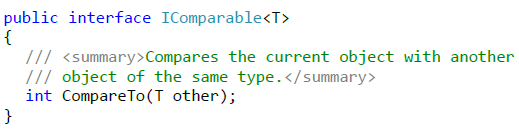
**Info:**

What does “**comparability between objects**” mean for us as developers? It means that we must somehow oblige everyone who uses our data structure, to create it passing it a **type, which is comparable**.

In C# the sentence “**type, which is comparable**” will sound like this:



The **interface IComparable<T>**, located in the namespace **System**, specifies the method **CompareTo(T obj)**, which returns a negative integer number, zero or a positive integer number respectively if the current object is less, equal or bigger than the one which is given to the method for comparing. Its definition looks approximately like this:



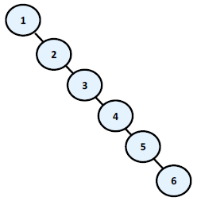
For the methods **Equals(Object obj)** and **GetHashCode()** a good (recommended) practice is these two methods to be consistent in their behavior, i.e. when two objects are the same, then their hash-code is the same.

**It’s recommended to sync the work of Equals(Object obj), CompareTo(T obj) and GetHashCode() methods. This is their expected behavior and it will save you a lot of hard to find problems!**

An important thing is the definition **BinarySearchTree<T> where T : IComparable<T>**. This constraint of the type **T** is necessary because of the requirement of our internal class, which works only with types, implementing **IComparable<T>**.

**Balanced Trees**

**Info:**

As we have seen above, the **ordered binary trees** are a very comfortable structure to search within. Defined in this way, the operations for creating and deleting the tree have a hidden flaw: **they don't balance the tree** and its depth could become very big.

Think a bit what will happen if we sequentially

include the elements: 1, 2, 3, 4, 5, 6? The ordered

binary tree will look like this:

In this case, the **binary tree degenerates**

**into a linked list**. Because of this the searching

in this tree is going to be much slower (with **N** steps, not with **log(N)**), as to check whether an item is inside, in the worst case we will have to go through all elements.

**Def:**

**Balanced binary tree** – a binary tree in which **no leaf is at “much greater” depth than any other leaf**. The definition of “much greater” is rough depends on the specific balancing scheme.

**Perfectly balanced binary tree** – binary tree in which the difference in **the left and right tree nodes’ count** of any node is at most one.

Without going in details we will mention that when given **binary search tree is balanced**, even not perfectly balanced, then the operations of **adding**, **searching** and **removing** an element in it will run in approximately a **logarithmic number of steps** even in the worst case.

These operations are called **rotations** in most of the cases. The type of rotation should be further specified and depends on the implementation of the specific data structure. As examples for structures like these we can give **Red-Black tree**, **AVL-tree**, **AA-tree**, **Splay-tree** and others.

**Balanced search trees** allow quickly (in general case for approximately **log(n)** number of steps) to perform the operations like **searching**, **adding** and **deleting** of elements.

This is due to two main reasons:

1. Balanced search trees keep their elements **ordered internally**.

2. Balanced search trees keep themselves **balanced**, i.e. their depth is always in order of **log(n)**.

Due to their importance in computer science we will talk about **balanced search trees** and their standard implementations in .NET Framework manytimes when we discuss data structures and their performance in this chapterand in the next few chapters.

Balanced search trees can be binary or non-binary. **Balanced binary search trees** have multiple implementations like **Red- Black Trees**, **AA Trees** and **AVL Trees**.

All of them are ordered, balanced and binary, so they perform insert / search / delete operations very fast.

**Non-binary balanced search trees** also have multiple implementations with different special properties. Examples are **B Trees**, **B+ Trees** and **Interval** **Trees**.

All of them are ordered, balanced, but not binary. Their nodes can typically hold more than one key and can have more than two child nodes. These trees also perform operations like insert / search / delete very fast.

**The Hidden Class TreeSet<T> in .NET Framework**

**Info:**

In the standard libraries of the .**NET Framework** there are ready implementations of **balanced trees**, but also on the Internet you can find a lot of **external libraries**.

In the namespace **System.Collections.Generic** a class **TreeSet<T>** exists, which is an **implementation of a red-black tree**.

The bad news is that this class is **internal** and it is visible only in this library. Fortunately, this class is used internally by a class, which is publicly available – **SortedDictionary<T>**.

**Graphs**

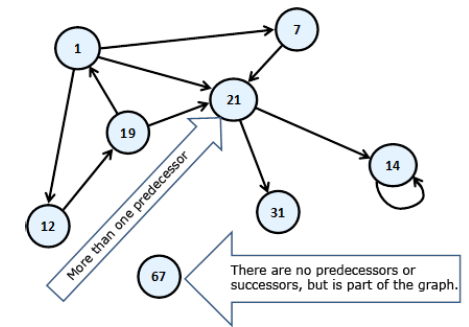
**Info:**

The **graphs** are very useful and fairly common data structures. They are used to describe a wide variety of **relationships between objects** and in practice can be related to almost everything.

Trees are a subset of the graphs and also lists are special cases of trees and thus of graphs, i.e. the graphs represent a generalized structure that allows modeling of very large set of real-world situations.

Frequent use of graphs in practice has led to extensive research in "**graph theory**", in which there is a large number of known problems for graphs and for most of them there are well-known solutions.

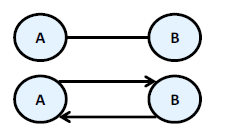
**Graphs – Basic Concepts**

**Info:**

**Finite directed graph** is called the couple (**V, E**), in which **V is a finite set** of **vertices** and **E** **is a finite set of directed edges**. Each edge ***e***that belongs to **E** is an ordered couple of vertices ***u***and ***v***or ***e*** *= (****u, v****)*, which are defining it in a unique way.

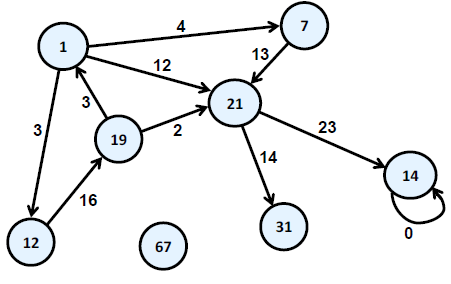
If instead of arrows, the vertices are connected with segments, then the segments will be called **undirected edges**, and the graph – **undirected**.

Practically we can imagine that an undirected edge from vertex A to vertex B is two-way edge and equivalent to two opposite directed edges between the same two vertices:



Two vertices connected with an edge are called **neighbors** (adjacent). For the edges a **weight function** can be assigned, that associates each edge to a real number. These numbers we will call **weights (costs)**.

For examples of the weights we can mention some distance between neighboring cities, or the length of the directed connections between two neighboring cities, or the crossing function of a pipe, etc. A graph that has weights on the edges is called **weighted**. Here is how it is illustrated a weighted graph.



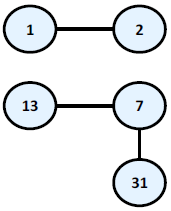
**Path in a graph** is a sequence of vertices v1, v2, …, vn,, such as there is an edge from vi to vi+1 for every i from 1 to n-1. In our example path is the sequence "1", "12", "19", "21". "7", "21" and "1" is not a path because there is no edge starting from "21" and ending in "1".

**Length of path** is the number of edges connecting vertices in the sequence of the vertices in the path. This number is equal to the number of vertices in the path minus one. The length of our example for path "1", "12", "19", "21" is three.

**Cost of path** in a weighted graph, we call the sum of the weights (costs) of the edges involved in the path. In real life the road from Sofia to Madrid, for example, is equal to the length of the road from Sofia to Paris plus the length of the road from Madrid to Paris. In our example, the length of the path "1", "12", "19" and "21" is equal to 3 + 16 + 2 = 21.

**Loop** is a path in which the initial and the final vertex of the path match. Example of vertices forming loop are "1", "12" and "19". In the same time "1", "7" and "21" do not form a loop.

**Looping edge** we will call an edge, which starts and ends in the same vertex. In our example the vertex "14" is looped.



A **connected undirected graph** we call an

undirected graph in which there is a path from each

node to each other. For example, the following

graph is **not connected** because there is no path

from "1" to "7".

So we already have enough knowledge to define the concept **tree in other way, as a special kind of graph**:

**Tree** – undirected connected graph without loops.

**Graphs – Presentations**

**Info:**

Without falling into greater details we will set out some of the most common representations of graphs.

**1. List of successors** – in this representation for each vertex **v** a list of successor vertices is kept (like the tree’s child nodes). Here again, if the graph is weighted, then to each element of the list of successors an additional field is added indicating the weight of the edge to it.

**2. Adjacency matrix** – the graph is represented as a square matrix **g[N][N]**, where if there is an edge from **vi** to **vj**, then the position **g[i][j]** is contains the value 1. If such an edge does not exist, the field **g[i][j]** is contains the value 0. If the graph is weighted, in the position **g[i][j]** we record weight of the edge, and matrix is called a **matrix of** **weights**. If between two nodes in this matrix there is no edge, then it is recorded a special value meaning infinity. If the graph is undirected, the adjacency matrix will be symmetrical.

**3. List of the edges** – it is represented through the list of ordered pairs (vi, vj), where there is an edge from **vi** to **vj**. If the graph is weighted, instead ordered pair we have ordered triple, and its third element shows what the weight of the edge is.

**4. Matrix of incidence between vertices and edges** – in this case, again we are using a matrix but with dimensions **g[M][N]**, where **N** is the number of vertices, and **M** is the number of edges. Each column represents one edge, and each row a vertex. Then the column corresponding to the edge **(vi, vj)** will contain 1 only at position **i** and position **j**, and other items in this column will contain 0. If the edge is a loop, i.e. is **(vi, vi)**, then on position **i** we record 2. If the graph we want to represent is oriented and we want to introduce edge from **vi** to **vj**, then to position **i** we write 1 and to the position **j** we write -1.

The most commonly used representation of graphs is the **list of successors**!

**Dictionaries, Hash-Tables and Sets**

**Dictionary Data Structure**

**Info:**

In the last few chapters we got familiar with some classic and very important data structures – **arrays**, **lists**, **trees** and **graphs**.

Data structure "**dictionaries"** are an extremely useful and widely used in the programming. The dictionaries are also known as **associative arrays** or **maps**.

Every element in the dictionary has a **key** and an **associated value** for this key. Both the key and the value represent a pair.

The analogy with the real world dictionary comes from the fact, that in every dictionary, for every for word (**key),** we also have a description related to this word (**value**).

**As well as the data (values), that the dictionary holds, there is also a key that is used for searching and finding the required values. The elements of the dictionary are represented by pairs (key, value), where the key is used for searching!**

**Info:**

When using the **ADT dictionary**, the key may not just be a number, but any other type of object.

In the case, when we have a key (number), we could implement this type of structure as a regular array.

In this scenario the **set of keys** is already known – these are the numbers from **0** to **n**, where **n** represents the size of the array (when **n** is within the allowed limits). The ideaof the dictionaries is to provide us with more flexibility regarding the set ofthe keys.

For every **key** in the **dictionary**, there is a corresponding **value**. One key can hold only one value. The aggregation of all the **pairs (key, value)** represents the dictionary.

**The Abstract Data Structure “Dictionary” (Associative Array, Map)**

**Info:**

In programming the **abstract data structure "dictionary"** is represented by many aggregated pairs (key, value) along with predefined methods for accessing the values by a given key.

Alternatively this data structure can also be called a "**map**" or "**associative array**".

Described below are the required operations, defined by this data structure:

- **void Add(K key, V value) –** adds given key-value pair in the dictionary. With most implementations of this class in .NET, when adding a key that already exists**, an exception is thrown**.

- **V Get(K key)** – returns the value by the specified key. If there is no pair with this key, the method returns **null** or throws an exception depending on the specific dictionary implementation.

- **bool Remove(key)** – removes the value, associated with the specified key and returns a Boolean value, indicating if the operation was successful.

Here are some additional methods, which are supported by the ADT:

- **bool Contains(key)** – returns **true** if the dictionary has a pair with the selected key;

- **int Count** – returns the number of elements (key value pairs) in the dictionary.

Other operations that are usually supported are: **extracting all of the keys**, **values or key value pairs and importing them into another structure** (**array**, **list**). This way they can easily be **traversed using a loop**.

**For the comfort of .NET developers, the IDictionary<K, V> interface holds an indexing property V this[K] { get; set; }, which is usually implemented by calling the methods V Get(K), Add(K, V).**

**Bear in mind that the access method (accessor) get of the property V this[K] of the class Dictionary<K, V> in .NET throws an exception if the given key K does not exist in the dictionary. In order to access the value of a certain key, without having to worry about exceptions, use the method bool TryGetValue(K key, out V value).**

**The Interface IDictionary<K, V>**

**Info:**

In .NET there is a standard interface **IDictionary<K, V>** where **K** defines the type of the **key**, and **V** type of the **value**.

**IDictionary<K, V>** corresponds to the abstract data structure "dictionary" and defines the operations, mentioned above, but without supplying an actual implementation of them.

This interface is defined in assembly **mscorelib**, **namespace System.Collections.Generic**.

In .NET **interfaces** represent **specifications of methods** for a certain class. They define methods without implementation, which should be implemented by the classes that inherit them.

**Interfaces define which methods and fields should be implemented in the classes that inherit the interface.**

In .NET Framework there are two major implementations of the interface **IDictionary<K, V>** – **Dictionary<K, V>** and **SortedDictionary<K, V>**.

**SortedDictionary** is an implementation by a balanced (red-black) tree, and **Dictionary** – by a hash-table.

**Except for IDictionary<K, V> in .NET there is one more interface – IDictionary, along with the classes implementing it: Hashtable, ListDictionary and HybridDictionary. They are heritage from the first version of .NET. These classes need to be used only on special occasions. Much more preferable is the use of Dictionary<K, V> or SortedDictionary<K, V>.**

**Implementation of Dictionary with Red-Black Tree**

**Info:**

A **red-black tree is an ordered binary balanced search tree**, that’s used for searching. This is why one of the important requirements for the set of keys used by **SortedDictionary<K, V>** is **comparability**. This means that, if we have two keys, either one of them should be bigger, or they should be equal.

The keys used in **SortedDictionary<K, V>** should implement **IComparable<K>**.

The usage of the **binary search tree** gives us a great advantage: the **keys in the dictionary are stored ordered**. Thanks to this feature, if we need the data ordered by keys, we don’t need to perform any additional sorting.

Actually, this is the only advantage of this dictionary implementation compared to the **hash-table**.

A thing that should be mentioned is that keeping the keys ordered comes with its price. Searching for the elements using in an **ordered balanced tree** is slower (typically takes **log(n)** steps) than using **a hash-table** (typical takes **fixed number of steps**). Because of this, if there is no requirement for the keys to be ordered, it’s better to use **Dictionary<K, V>**.

**Use a balanced tree dictionary only when you need your pairs (key, value) to be ordered by key. Bear in mind that the balanced tree comes with the complexity of the algorithm log(n), for searching, adding and deleting elements. Compared to this, the complexity used in hash-table may reach a linear value!**

**The Class SortedDictionary<K, V>**

**Info:**

The class **SortedDictionary<K, V>** is a dictionary implementation, which uses a **red-black tree**. This class implements all the standard operations defined in the interface **IDictionary<K, V>**.

**IComparable<K> Interface**

**Info:**

When using **SortedDictionary<K, V>** the keys are required to be **comparable**. In our example we use objects of type **string**.

The class **string** implements the interface **IComparable**, and the comparison between the elements is done lexicographically. What does that mean? By default the strings in .NET are case sensitive (the compiler distinguishes uppercase from lowercase letters). Words like "Length" and "length" are considered different. This means that words that start with a lowercase letter will be before the ones with an uppercase letter. This definition comes from the implementation of the method **CompareTo(object)**, through which the **string** class implements the interface **IComparable**.

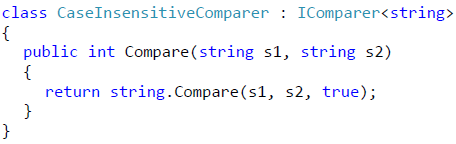
**IComparer<T> Interface**

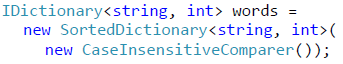
**Info:**

For the comparison of objects with an exclusively defined order in **SortedDictionary<K, V>** in .NET, we will use the interface **IComparer<T>**. It defines a comparison function **int Compare(T x, T y)** that is an alternative to the already defined order. Let’s take a better look at this interface.

When we create an object of type **SortedDictionary<K, V>** we can pass to its constructor a reference to **IComparer<K>** so that it can use it for the key comparison (key elements should be objects of type **K**).

Here is a sample implementation of **IComparer<K>** that changes the behavior when comparing strings, so that they are **not** distinguished by uppercase and lowercase characters:



Let’s use this interface **IComparer<E>** when creating the dictionary:

**When two objects are equal (Equals(object) returns true), CompareTo(E) should return 0!**

**Hash-Tables**

**Info:**

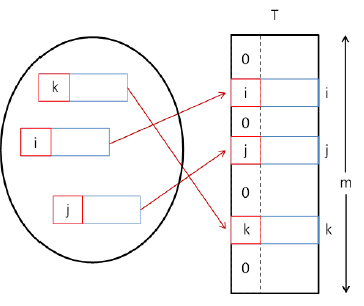
The data structure **hash-table**, which implements the abstract data structure **dictionary** in a **very efficient way**.

**Dictionary Implementation with Hash-Table**

**Info:**

With a **hash-table implementation**, the time for accessing the elements in the dictionary is theoretically **independent from their count**. This is a very important advantage.

The data structure **hash-table** is usually implemented internally with an **array**. It consists of **numerated elements** (cells), each either **holding a key-value pair** or is **empty** (**null**). The figure below illustrates how a hash-table might look like:



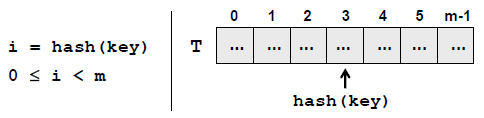
The size of the internal storage array of the hash-table is called **capacity**. The **load factor** is a real number between 0 and 1, which stands for the ratio between the occupied elements and the current capacity.

At the figure we have a hash-table with **3** elements and capacity **m**. The load factor for this hash-table would be **3/m**.

When adding or searching for elements, a method for hashing the key (**hash function**) is executed **hash(key)**, that returns a number we call a **hash-code**. When we take the division remainder of this hash-code and the capacity **m** we get a number between **0** and **m-1**:



At the figure there is a hash-table **T** with capacity **m** and hash-function **hash(key)**:



This value **hash(k)** gives us the **position** in the array at which we search or add a certain **key-value pair** having this **k**. If the hash-function distributes the keys uniformly, in most cases for every key a different hash value will be assigned. In this way **every cell of the array will have at most one key**.

Ultimately we get an extremely fast search and insertion of the elements: just **calculate the hash function and obtain the cell assigned for the key**. Of course it may occur that different keys would have the same hash code. We will examine this special case in more details later.

**Use implementation of dictionary based on hash-table, when you need to find values by key with a maximum speed!**

The internal table’s **capacity is increased** when the number of elements in the hash-table becomes greater or equal to a certain constant called **fill factor** (load factor, the maximal degree of filling). When increasing the capacity (usually doubling it), all of the elements are reordered by the hash code of their keys and their assigned cell is calculated according to the new capacity.

The **load factor** is significantly decreased after the reordering. This operation is time-consuming, but it is executed relatively rare, so it will not impact the overall performance of the "add" operation.

**Class Dictionary <K, V>**

**Info:**

The class **Dictionary<K, V>** is a standard implementation of a **dictionary based on hash-table** in .NET Framework.

**Class Dictionary<K, V> – Main Operations**

**Info:**

Creating a hash-table is done by calling some of the constructors of **Dictionary<K, V>**.

It’s good if we know in advance the expected number of elements, which would be added in our hash-table, so as to set it at the creation of the hash-table. This way we will avoid the unneeded expansions of the hash-table and we will achieve better performance.

By default the value of the **initial capacity is 16**, and the **load factor is 0.75**.

Let’s review the methods in the class **Dictionary<K, V>**:

**1. void Add(K, V)** adds a new pair (key and a value) to the hash-table. Throws an exception in the case that the key exists. This operation is extremely fast.

**2. bool TryGetValue(K, out V)** returns an element of type V via the **out** parameter for the given key or **null**, if there is no such key. The result of this operation will be **true** if such an element is found. The operation is very fast, because the algorithm for searching an element by key in the hash-table is with complexity about O(1)

**3. bool Remove(K)** removes the element with this key. This operation works very fast.

**4. void Clear()** removes all the elements from the dictionary.

**5. bool ContainsKey(K)** check if there is an ordered pair with this key in the dictionary. This operation works extremely fast.

**6. bool ContainsValue(V)** checks if there is one or more ordered pairs with this value. This operation is slow because it checks every element of the hash-table (like searching in a list).

**7. int Count** returns the number of ordered pairs within the dictionary.

**8.** Other operations – extracting all the keys, values or ordered pairs into a structure that could be iterated through using a loop.

**Info:**

In hash-tables (unlike balanced trees) the elements **are not kept sorted**. Even if the current table capacity is changed while working with it, it is also highly possible that the order of the pairs could be changed as well.

It is important to remember, that with hash-tables, we cannot rely on the elements being in order. If we need them ordered, we could sort the elements before printing. Another option would be using **SortedDictionary<K, V>**.

**Hashing and Hash-Functions**

**Info:**

The **hash-code** is a number returned by the **hash-function**, used for the **hashing the key**. This number should be different for every key, or at least there should be a high chance for that.

**Hash-Functions**

**Info:**

There is the concept of the **perfect hash-function**. One hash-function is called **perfect**, if for example you have N keys, and for each of them the function would add a **different number** in a reasonable interval (for example from 0 to N-1).

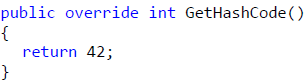
Finding such a function in the common case is a very hard, **almost impossible** task. In practice there are also other, **not so "perfect" hash-functions**.

The are few examples for hash-functions, which are used directly with .NET libraries:

**The Method GetHashCode() in .NET Framework**

**Info:**

Every .NET class has a method called **GetHashCode()** that returns a value of type **int**. This method is inherited by the class **Object**, which is the root member in the hierarchy of .NET classes.

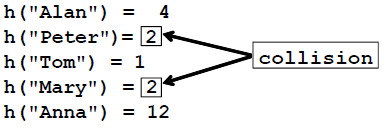
 The implementation in the class **Object** of the method **GetHashCode() does not guarantee** the unique value of the result. This means that thedescendent classes need to ensure that **GetHashCode()** is implemented inorder to use it for a key in a hash-table.

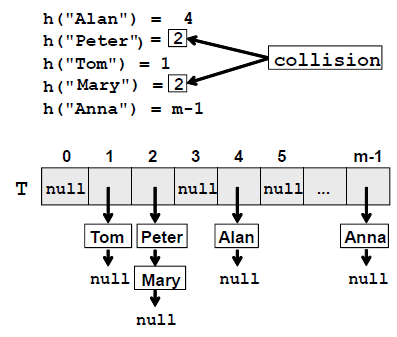
If in a hash-table we use objects for keys from a class, that has the above implementation of **GetHashCode()**, it will have **very poor performance**, because every time, when we add a new element in the table, we would have to insert it **at the same place**. Every time we search the hash-table, we will encounter the same element.

In order to avoid the described behavior, we need the hash-function to **distribute the keys evenly** amongst the possible hash-code values.

**Collisions with Hash-Functions**

**Info:**

 The situation where **two different keys have the same hash-code** is called **collision**. A good example of collision is shown below:

 The simplest solution is obvious: order the pairs that have keys with the same hash-codes in a **list** or other data structure. Thus we don't solve the collisions but we accept them and we just put several key-value-pairs in the same element in the underlying array in the hash-table. This approach for collision resolution is known as chaining:

Therefore when using a constant

42 for hash-code our hash-table turns into

a linear list and it becomes very inefficient.

**Implementing the Method GetHashCode()**

**Info:**

A standard algorithm for implementing **GetHashCode()**, when this is necessarily:

**1.** First we need to choose which fields of the class will take part in the implementation of the **Equals(object)** method. This is necessary, because every time when **Equals()** returns **true**, the result from **GetHashCode()** should always return the same value.

This way the fields that do not take part in **Equals()**, should not take part in **GetHashCode()** as well.

**2.** After we choose which fields will take part for the calculation of **GetHashCode()**, we need to receive values from them (of type **int**). Here is a sample scheme:

- **If** the field is **bool**, for **true** we take **1**, and for **false** we take **0** (or directly call method **GetHashCode()** on **bool**).

- **If** the field is of type **int**, **byte**, **short**, **char**, we can convert it to **int**, with the cast operator (**int**) (or we could directly call **GetHashCode()**).

- **If** the field is type **long**, **float** or **double**, we could use the result from their own implementations of **GetHashCode()**.

- **If** the field is not a primitive type, we could call the method **GetHashCode()** of this object. If the field value is **null**, we can return **0**.

**- If the field is an array or a collection, we take the hash-code from every element of this collection!**

In the end we sum all the received **int** values, and before each addition we multiply the temporary result with a prime number (for example 83), while ignoring the eventual overflow of type **int**. For example, if we have 3 fields and their hash codes are **f1**, **f2** and **f3**, our hash function could combine them though the formula:

At the end we obtain a hash-code, **hash**=(((**f1\*83**) **+ f2**)**\*83)**+**f3**.

which is very well distributed in the

range of all 32-bit values. We can expect, that with a hash-code calculated this way, the **collisions would be rare**, because every change in some of the fields taking part in **GetHashCode()** leads to a major change in the hash code and thus reduces the chance for collision.

**Interface IEqualityComparer<T>**

**Info:**

One of the most important things that we have learned so far is that in order to use instances of a class as keys for a dictionary, the class needs to properly implement **GetHashCode()** and **Equals(…)**.

But what should we do if we want to use a class, that we cannot inherit or change? In this case the interface **IEqualityComparer<T>** comes to our aid.

It defines the following two operations:

1. **bool Equals(T obj1, T obj2)** – returns **true** if **obj1** and **obj2** are Equal;

2. **int GetHashCode(T obj)** – returns the hash-code of given object.

**The dictionary uses interface methods instead of the corresponding methods of the class that is used for key!**

**Remember that the keys in hash-tables need to have correctly defined Equals(…) and GetHashCode() to work properly!!**

**This is not required for the values, just for the keys!**

**Always define both Equals(…) and GetHashCode(), never only one of them!**

**Resolving the Collision Problem**

**Info:**

In practice, **collisions happen almost always**, excluding some rare and specific cases. That is why we need to live with the idea of the collisions presence in our hash-tables and take them into account.

**Chaining in a List**

**Info:**

The most widespread **method to resolve collisions** problem is called **chaining**. Its major concept consists of storing in a list all the pairs (key, value), which have the same hash-code for the key.

**Open Addressing Methods for Collision Resolution**

**Info:**

Now let’s look over the **methods for collision resolution**, alternative to chaining in a list. In general, the idea is, in case of collision we try to put the new pair in a table position, which is free.

**Linear Probing**

**Info:**

This is one of easiest methods for implementation. **Linear probing**, in general, can be presented with the following sample code:



Here **capacity** is the internal table capacity, **oldPostion** is the position where collision occurs and **i** is a number for the next probing. If the new position is free, then we place the new pair there. Otherwise we try again (probing), incrementing **i**. Probing can be either forward or backwards. Backward probing is when instead of adding, we are subtracting **i** from the position we have collision for.

The advantage of this method is the **relatively quick way to find of a new position**. Unfortunately, if there was a collision at a certain place, there is an extremely high probability collision to occur again at the same place. So this, in practice, leads to a high inefficiency.

**Using linear probing as a method for collision resolution in hash tables is inefficient and has to be avoided!**

**Quadratic Probing**

**Info:**

Quadratic probing is a classic method for collision resolution. The main difference between **quadratic probing** and **linear probing** is that it uses a quadratic function of **i** (the number of the next probing) to find new position.

Possible quadratic probing function is shown below:



The given example uses two constants: **c1** and **c2**, such that **c2** must not be **0**, otherwise we are going back to linear probing.

By choosing **c1** and **c2** we define the position we are going to probe, compared to the starting position. For instance, if **c1** and **c2** are equal to **1**, we are going to probe consequently **oldPosition**, **oldPosition + 2**, **oldPosition + 6**, …

For a hash-table with capacity of the kind **2n**, the best is to choose **c1** and **c2** equal to **0.5**.

**Quadratic probing is more efficient than linear the linear probing.**

**Double Hashing**

**Info:**

As the name implies, the **double hashing** method uses **two different hash functions**.

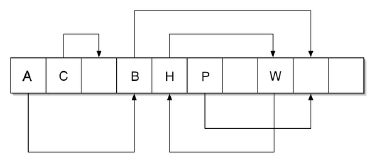
The main concept is that, the second hash function is used for the elements that fall into a collision. This method is better than the linear and quadratic probing, because all the next probing depends of the value of the key and not of the table position inside the hash-table. It makes sense, because the position of a given key depends on the current capacity of the hash-table.

**Chuckoo Hashing**

**Info:**

**Cuckoo hashing** is a relatively new method for collision resolution, using an open addressing. Its name comes from the behavior, observed with some kinds of cuckoos. The mother cuckoos push out the eggs and/or the nests out of other birds, in order to put their own eggs there and the other birds mistakenly care for the cuckoos' eggs in that way. (Also for the nests, after the incubation).

The main idea of this method is the use of **two hash-functions** instead of one. In this way, we have **not one, but two positions to place the** **element** inside the hash-table. If one of the positions is free, then we just put the element there. If both are taken, then we put the new element in one of them and it "**kicks out**" the element, which was already there. In turn, the "kicked" element is going to his alternative position and "kicks" another element out, if necessary. The new "kicked out" is repeating the procedure, and in that way until reaching a free position or we fall into a loop. In the last case, the whole hash table is built again with greater size and new hashfunctions.



Let’s assume that this is the cell, containing element A. The new element "kicks out" A from his place, A in turn goes to its alternative position and "kicks out" B from his place. The **alternative position** of B is free, so the adding is successfully completed.

Let’s assume, that the cell, the new element is trying to "**kick out**" an element, is the cell containing H. Then we have a loop, formed by H and W. In this case, a rebuild must be done using greater size, and new hash-functions.

Some researches show, the **cuckoo hashing and its modifications could be much more efficient than the widely spread today chaining in a list** and open addressing methods.

Nevertheless, this method is still not well adopted in the industry and not used internally in .NET Framework.

The main stopper is the need of **two hash func**tions, which means that the class **System.Object** should introduce two **GetHashCode()** methods.

**The "Set" Data Structure**

**The Abstract Data Structure "Set"**

**Info:**

**Sets** are **collections of unique elements** (without any repeating elements inside).

In the .NET context, it means, for every set object, calling its **Equals()** method and passing another object from the set as an argument, will always result in **false**.

Note that two different objects in .NET **may be equal when compared by certain field** and thus in the data structure "set" only one of them could be put.

**Some sets allow their elements to be null, while others do not allow.**

Besides not allowing the repetition of objects, another important thing, that distinguishes sets from lists and arrays, is that **the set element has no index.**

The elements of the set cannot be accessed by any key, as it is with dictionaries. **The elements themselves are the keys.**

**The only way to access an object from a set is by having available either the object itself or another object, which is equal to it. That is why, in practice we access all the elements of a given set at once, while iterating, by using the foreach loop construct.**

**Set Implementations in .NET Framework**

**Info:**

In .NET (version 4.0 and above) there is an **interface ISet<T> representing the ADT "set"** and it has two standard implementation classes:

**- HashSet<T> – hash-table** based implementation of set.

**- SortedSet<T>** – **red-black tree** based implementation of set.

The main operations, defined by the **ISet<T>** interface (abstract data structure set), are the following:

- **bool Add(element)** – adding the **element** to the set and returning **false** if the element is already present inside the set, otherwise returning **true**.

- **bool Contains(element)** – checks if the set already contains the element passed as an argument. If yes, returns **true** as a result, otherwise returns **false**.

- **bool Remove(element)** – removes the **element** from the set. Returns Boolean if the element has been present inside the set.

- **void Clear()** – removes all the elements from the set.

- **void IntersectWith(Set other)** – inside the current set remain only the elements of the intersection of both sets – the result is a set, containing the elements, which are present in both sets at the same time – the set, calling the method and the **other**, passed as parameter.

- **void UnionWith(Set other)** – inside the current set remain only the elements of the sets union – the result is a set, containing the elements of either one or the other, or both sets.

- **bool IsSubsetOf(Set other)** – checks if the current set is a subset of the **other** set. Returns **true**, if yes and **false**, if no.

- **bool IsSupersetOf(Set other)** – checks if the **other** set is a subset of the current one. Returns **true**, if yes and **false**, if no.

- **int Count** – a property, which returns the current number of elements inside the set.

**Data Structures and Algorithm Complexity**

**Info:**

**Data structures and algorithms are the fundamentals of programming. In order to become a good developer it is essential to master the basic data structures and algorithms and learn to apply them in the right way!**

**Algorithm Complexity**

**Info:**

We cannot talk about **efficiency of algorithms and data structures** without explaining the term "algorithm complexity", which we have already mentioned several times in one form or another.

**Algorithm complexity** is a **measure** which evaluates the order of the **count** **of operations**, performed by a given or algorithm as a function of the size of the input data. To put this simpler, complexity is a rough **approximation of the number of steps** necessary to execute an algorithm.

When we evaluate **complexity** we speak of **order of operation count**, not of their exact count. For example if we have an order of N^2 operations to process N elements, then N^2/2 and 3\*N^2 **are of one and the same quadratic order**.

**Algorithm complexity** is commonly represented with the **O(f)** notation, also known as **asymptotic notation** or “**Big O notation**”, where **f** is the function of the size of the input data.

The **asymptotic computational** complexity **O(f)** **measures the order of the consumed resources** (**CPU time**, **memory**, etc.) by certain algorithm **expressed as function of the input data size**.

Complexity can be **constant**, **logarithmic**, **linear**, **n\*log(n)**, **quadratic**, **cubic**, **exponential**, etc.

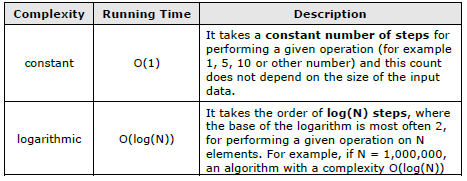
This is respectively the order of constant, logarithmic, linear and so on, number of steps, are executed to solve a given problem. For simplicity, sometime instead of “**algorithms complexity**” or just “**complexity**” we use the term “**running time**”.

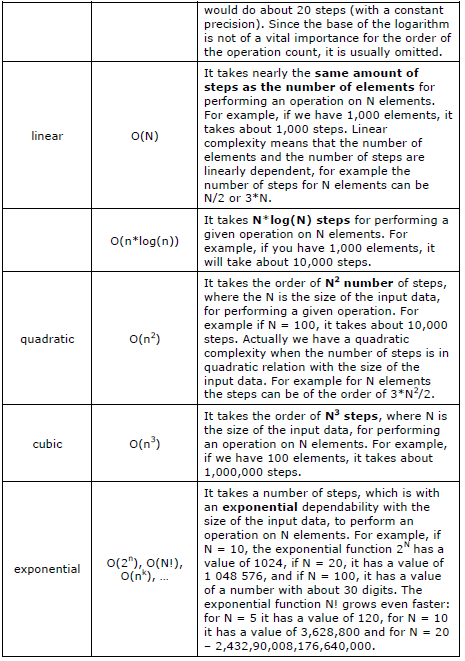
**Algorithm complexity is a rough approximation of the number of steps, which will be executed depending on the size of the input data. Complexity gives the order of steps count, not their exact count.**

**Typical Algorithm Complexities**

**Info:**

This table will explain what every type of complexity (running time) means:





When evaluating complexity, **constants are not taken into account**, because they do not significantly affect the count of operations. Therefore an algorithm which does N steps and algorithms which do N/2 or 3\*N respectively are considered linear and approximately equally efficient, because they perform a number of operations which is of the same order.

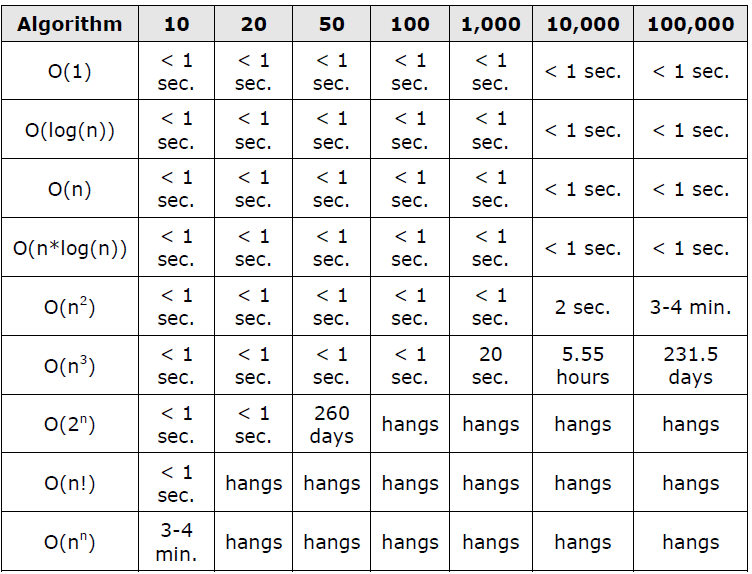
**Complexity and Execution Time**

**Info:**

The **execution speed** of a program depends on the complexity of the algorithm, which is executed.

If this complexity is low, the program will execute fast even for a big number of elements.

If the complexity is high, the program will execute slowly or will not even work (it will hang) for a big number of elements.



We can draw many **conclusions** from the above table:

- Algorithms with a **constant**, **logarithmic** or **linear complexity** are so **fast** that we cannot feel any delay, even with a relatively big size of the input data.

- Complexity **O(n\*log(n))** is **similar to the linear** and works nearly as fast as linear, so it will be very difficult to feel any delay.

- **Quadratic** algorithms work very well up to several thousand elements.

- **Cubic** algorithms work well if the elements are not more than 1,000.

- Generally these so called **polynomial algorithms** (any, which are not exponential) are considered to be fast and working well for thousands of elements.

- Generally the **exponential algorithms do not work well** and we should avoid them (when possible). If we have an exponential solution to a task, maybe we actually do not have a solution, because it will work only if the number of the elements is below 10-20. Modern cryptography is based exactly on this – there are not any fast (non-exponential) algorithms for finding the secret keys used for data encryption.

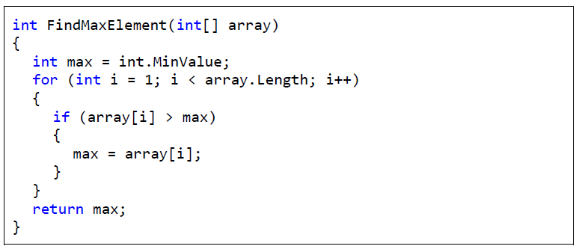
**If you solve a given problem with an exponential complexity this means that you have solved it for a small amount of input data and generally your solution does not work!**

**Best, Worst and Average Case**

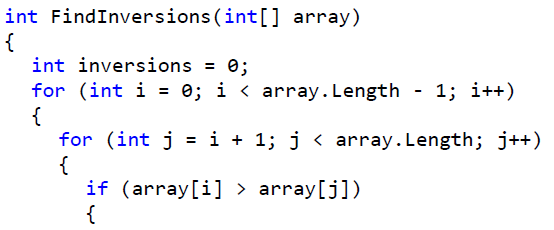
**Info:**

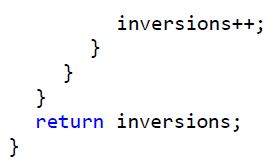
Complexity of algorithms is usually evaluated in the **worst case** (most unfavorable scenario). This means in the average case they can work faster, but in the worst case they work with the evaluated complexity and not slower.

**Estimating Complexity – Examples**

If we have a single loop from 1 to N, its complexity is **linear – O(N):**

If we have **two of nested loops from 1 to N**, their complexity is **quadratic – O(N2):**



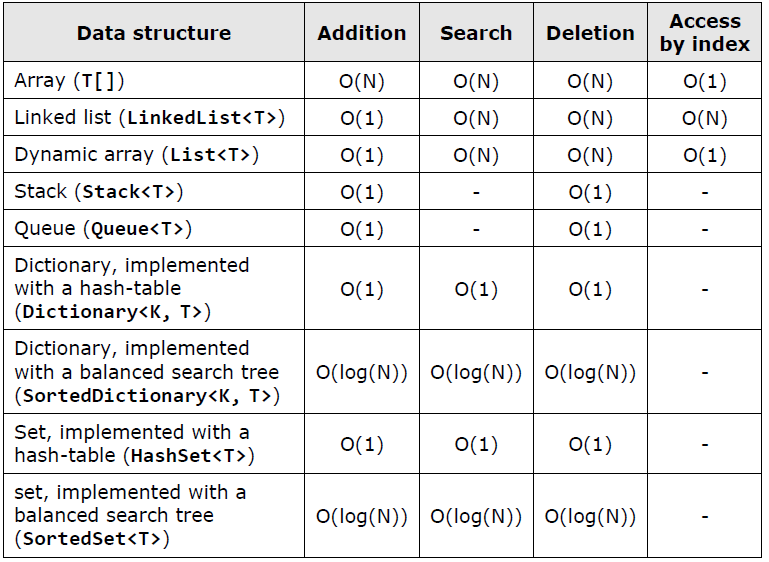


**Comparison between Basic Data Structures**

**Info:**

We are now ready to make a **comparison between the basic data structures**, which we know from the last few chapters, and to estimate with what complexity each of them performs the basic operations like **addition**, **searching**, **deletion** and **access by index** (when applicable).

In that way we could easily judge according to the operations we expect to need, **which structure** **would be the most appropriate**. The complexities of the basic operations on the basic data structures, which we have reviewed in the previous chapters, are given in the table below:



**When to Use a Particular Data Structure?**

**1. Array (T[])**

**Info:**

The arrays are **collections of fixed number of elements** from a given type (for example numbers) where the **elements preserved their order**. Each element can be accessed through its index. The arrays are memory areas, which have a **predefined size**.

**Adding a new element** in an array is a **slow operation**. To do this we have to allocate a memory with the same size plus one and copy all the data from the original array to the new one.

**Searching** in an array **takes time** because we have to compare every element to the searched value. It takes N/2 comparisons in the average case.

**Removing an element** from an array is a **slow operation**. We have to allocate a memory with the same size minus one and copy all the old elements except the removed one.

**Accessing by index** is direct, and thus, a **fast operation**.

The arrays should be used only when we have to p**rocess a fixed number of elements** to which we need a **quick access by index**.

For example, if we have to sort some numbers, we can keep them in an array and then apply some of the well-known sorting algorithms. If we have to change the elements' count, the array is not the correct data structure we should use.

**Use arrays when you have to process a fixed number of elements to which you need an access through index!**

**2. Singly / Doubly Linked List (LinkedList<T>)**

**Info:**

Singly and doubly **linked lists** hold **collection of elements**, which preserve their order.

Their representation in the memory is dynamic, pointer-based. They are linked sequences of element.

**Adding is a fast operation** but it is a bit slower than adding to a **List<T>** because every time when we add an element to a linked list we allocate a new memory area.

The memory allocation works at speed, which cannot be easily predicted.

**Searching** in a linked list is a **slow operation** because we have to traverse through all of its elements.

**Accessing an element by index** is a slow operation because there is **no indexing** in singly and doubly linked lists. You have to go through all the elements from the start one by one instead.

**Removing** an element at a specified index is a **slow operation** because reaching the element through its index is a slow operation.

Removing an element with a specified value is a slow operation too, because it involves searching.

Linked list can **quickly add and remove elements** (with a constant complexity) **at its two ends** (head and tail). **Hence, it is very handy for an implementation of stacks, queues and similar data structures.**

Linked lists are **rarely used** in practice because the dynamic arrays (**List<T>**) can do almost exact same operations **LinkedList** does, plus for the most of them it works faster and more comfortable.

When you need a linked list, use **List<T>** instead of **LinkedList<T>**, because it doesn’t work slower and it gives you better speed and flexibility. Use **LinkedList** when you have to **add and remove elements at both ends** of the data structure.

**When you need to add and remove elements at both ends of the list, use LinkedList<T>. Otherwise use List<T>!**

**3. Dynamic Array (List<T>)**

**Info:**

Dynamic array (**List<T>**) is one of the **most popular data structures** used in programming.

It does not have fixed size like arrays, and allows direct access through index, unlike linked lists (**LinkedList<T>**).

The dynamic array is also known as "array list", "resizable array" and "dynamic array".

**List<T>** holds its elements in an array, which has a bigger size than the count of the stored elements. Usually when we **add an element**, there is an empty cell in the list’s inner array.

Therefore this operation takes a **constant time**. Occasionally the array has been filled and it has to expand. This takes linear time, but it rarely happens.

If we have a large amount of additions, the average-case complexity of adding an element to **List<T>** will a constant – O(1). If we sum the steps needed for adding 100,000 elements (for both cases – "fast add" and "add with expand") and divide by 100,000, we will obtain a constant which will be nearly the same like for adding 1,000,000 elements.

This statistically-averaged complexity calculated for large enough amount of operations is called **amortized complexity**.

Amortized linear complexity means that if we add 10,000 elements consecutively, the overall count of steps will be of the order of 10,000.

In most cases add it will execute in a constant time, while very rarely adding will execute in linear time.

Searching in **List<T>** is a **slow operation** because you have to traverse through all the elements.

**Removing by index** or value executes in a linear time. It is a **slow operation** because we have to move all the elements after the deleted one with one position to the left.

The **indexed access** in **List<T>** is instant, in a **constant time**, since the elements are internally stored in an array.

Practically **List<T> combines the best of arrays and lists**, for which it is a preferred data structure in many situations. For example if we have to process a text file and to extract from it all words (with duplicates), which match a regular expression, the most suitable data structure in which we can accumulate them is **List<T>**, because we need a list, the length of which is unknown in advance and can grow dynamically.

The dynamic array (**List<T>**) is appropriate, when we have to add elements frequently as well as keeping their order of addition and access them through index.

If we often we have to search or delete elements, **List<T>** is not the right data structure.

**Use List<T>, when you have to add elements quickly and access them through index!**

**4. Stack**

**Info:**

**Stack** is a linear data structure in which there are 3 operations defined:

- adding an element at the top of the stack (**push**),

- removing an element from the top of the stack (**pop**)

- inspect the element from the top without removing it (**peek**).

All these operations are **very fast** – it takes a **constant time** to execute them. The stack does not support the operations search and access through index.

The stack is a data structure, which has a **LIFO behavior** (last in, first out). It is used when we have to model such a behavior – for example, if we have to keep the path to the current position in a recursive search.

**Use a stack when you have to implement the behavior "last in, first out" (LIFO)!**

**5. Queue**

**Info:**

**Queue** is a linear data structure in which there are two operations defined:

**1.** adding an element to the tail (**enqueue**)

**2.** extract the front-positioned element from the head (**dequeue**).

These two operations take a **constant time** to execute, because the queue is usually implemented with a linked list. **The linked list can quickly add and remove elements from its both ends.**

The queue’s behavior is **FIFO (first in, first out)**. The operations searching and accessing through index are not supported.

Queue can naturally model a list of waiting people, tasks or other objects, which have to be processed in the same order as they were added (enqueued).

As an example of **using a queue** we can point out the implementation of the **BFS (breadth-first search) algorithm**, in which we start from an initial element and all its neighbors are added to a queue.

After that they are processed in the order they were added and their neighbors are added to the queue too. This operation is repeated until we reach the element we are looking for or we process all elements.

**Use a queue when you have to implement the behavior "first in, first out" (FIFO)!**

**6. Dictionary, Implemented with a Hash-Table (Dictionary<K, T>)**

**Info:**

The data structure "**dictionary**" suggests **storing key-value pairs** and provides a **quick search by key**.

The implementation with a hash table (the class **Dictionary<K,T>** in .NET Framework) has a **very fast add, search and remove** of elements – **constant complexity** at the average case.

The operation access through index is not available, because the elements in the hash-table have no order, i.e. an almost **random order**.

**Dictionary<K,T>** keeps internally the elements in an **array** and puts every element at the position calculated by the hash-function.

Thus the array is partially filled – in some cells there is a value, others are empty.

If more than one element should be placed in a single cell, elements are stored in a linked list. It is called **chaining**.

**This is one of the few ways to resolve the collision problem.** When the load factor exceeds 75%, the size is doubled and all the elements occupy new positions.

This operation has a linear complexity, but it is executed so rarely, that the amortized complexity remains a constant.

Hash-table has one peculiarity: if we choose a **bad hash-function** causing many collisions, the basic operations can become **very inefficient** and reach linear complexity.

In practice, however, this hardly happens. Hash-table is considered to be the fastest data structure, which provides adding and searching by key.

Hash-table in .NET Framework permits **each key to be put only once**. If we add two elements with the same key consecutively, the last will replace the first and we will eventually lose an element.

**This important feature should be considered.**

From time to time one key will have to keep multiple values. This is not standardly supported but we can store the values matching this key in a **List<T>** as a sequence of elements.

For example if we need a hash-table **Dictionary<int, string>**, in which to accumulate pairs {integer, string} with duplicates, we can use **Dictionary<int, List<string>>**.

Some external libraries have ready to use data structure called **MultiDictionary<K,V>**.

Hash-table is **recommended** to be used every time we need **fast addition** and **fast search by key**.

For example if we have to count how many times each word is encountered in a set of words in a text file, we can use **Dictionary<string, int>** – the key will be a particular word, the value – how many times we have seen it.

**Use a hash-table, when you want to add and search by key very fast!**

A lot of programmers (mostly beginners) live with the delusion the main advantage of using a hash-table is the comfort of **searching a value by its key**. Actually this is wrong.

We can implement searching a key with an array, a list or even a stack. There is no problem, everyone can build it.

We can define a class **Entry**, which holds a key-value pair and after that we will work with an array or a list with **Entry** elements.

We can implement the search but by any circumstances it will work slowly. This is the big problem with lists and arrays – they do not offer a fast search.

Unlike them the hash-table can **search and add new elements very fast**.

**The main advantage of the hash-table over the other data structures is a very quick searching and addition. The comfort for the developers is a secondary factor!**

**7. Dictionary, Implemented with a Balanced Tree (SortedDictionary<K, T>)**

**Info:**

The implementation of the data structure **"dictionary" as a red-black tree** (the class **SortedDictionary<K,T>**) is a structure storing key-value pairs where keys are ordered increasingly (sorted).

The structure provides a **fast execution** of basic operations (add an element, search by key and remove an element).

The complexity of these operations is **logarithmic** – O(log(N)). Thus, it will take 10 steps for add / search / remove when the dictionary holds 1,000 elements and 20 steps in case of 1,000,000 elements.

Unlike hash-tables, where we can reach linear complexity if we pick a bad hash-function, in **SortedDictionary<K,T>** the count of the steps of the basic operations in the average and worst case are the same – **log2(N)**.

**When we work with balanced trees, there is no hashing, no collisions and no risk of using a bad hash-function.**

Again, as in the hash-tables, one key can be stored at most once in the structure. If we want to associate several values with one key, we should use some kind of a list for the values, for example **List<T>**.

**SortedDictionary<K,T>** holds internally its values in a **red-black balanced tree** ordered by key. This means if we traverse the structure (using its iterator or **foreach** loop in C#) we will get the elements sorted in ascending order by key. **Sometimes this can be very useful property.**

Use **SortedDictionary<K,T>** when you need a structure which can **add, search and remove an element fast and you also need to extract the elements sorted in ascending order**. In general **Dictionary<K,T>** works a bit faster than **SortedDictionary<K,T>** and is preferable.

As an example of using a **SortedDictionary<K,T>**, we can give the following task: **find all the words in a text file, which occur exactly 10 times, and print them alphabetically**.

This is a task that we can solve as successful with **Dictionary<K,T>** too, but we will have to do an additional sorting at the end.

For the solution of this task we can use **SortedDictionary<string, int>** and to traverse through all the words in the text file. For each word we will keep in the sorted dictionary how many times we have encountered it.

After that we can go through all the elements in the dictionary and print those words, which have been encountered exactly 10 times. They will be alphabetically ordered, since this is the natural internal order of the sorted dictionary data structure.

**Use SortedDictionary<K,T> when you want fast addition of elements and searching by key as well as the elements to be sorted by key!**

**8. Set, Implemented with a Hash-Table (HashSet<T>)**

**Info:**

The data structure **"set" is a collection of elements with no duplicates**. The basic operations are adding an element to the set, checking if an element belongs to the set (searching) and removing an element from the set.

The operation searching through index is not supported, i.e. we do not have a direct access to the elements via ordering number, because **in this structure there is not any order**.

Set, implemented with a **hash-table** (the class **HashSet<T>**) is a special case of a hash-table, in which we have only keys. The values associated with these keys do not matter.

As in the hash-table, the basic operations in the data structure **HashSet<T>** are implemented with a **constant complexity O(1)**. Another similarity to hash-table is if we choose a bad hash-function, we can reach a linear complexity executing the basic operations. **Fortunately in practice this almost never happens.**

As an example of using a **HashSet<T>**, we can point out the task of finding all the different words in a text file.

**Use HashSet<T>, when you have to quickly add elements to a set and check whether a given element belongs to a set!**

**9. Set, Implemented with a Balanced Tree (SortedSet<T>)**

**Info:**

The data structure set, implemented with a red-black tree, is a special case of **SortedDictionary<K,T>** in which keys and values coincide.

Similar to **SortedDictionary<K,T>**, the basic operations in **SortedSet<T>** are executed with logarithmic complexity **O(log(N))**, which is the same in the average and worst case.

As an example of using a **SortedSet<T>** we can point out the task of finding all the different words in a given text file and printing them alphabetically ordered.

**Use SortedSet<T>, when you have to quickly add an element to a set and check whether given element belongs to the set as well as need all the elements sorted in ascending order!**

**Choosing a Data Structure – Conclusions**

**Info:**

By the many examples it is clear that the **choice of an appropriate data structure is highly dependable on the specific task**. Sometimes **data structures have to be combined** or we have to use several of them simultaneously.

What data structure should we pick mostly **depends on the operations we will perform**, so always ask yourselves "what operations should the structure, I need, perform efficiently".

If you are familiar with the operations, you can easily conform which structure does them most efficiently and at the same time is easy and handy.

In order to efficiently choose an appropriate data structure, you should firstly **invent the algorithm**, which you are going to implement, and then **look for an appropriate data structures for it.**

**Always go from the algorithm to the data structures, never backwards.**

**External Libraries with .NET Collections**

**Info:**

It is a well-known fact that the standard data structures in **.NET Framework System.Collections.Generic** have pretty poor functionality. It lacks implementations of basic concepts in data structures such as multi-sets, priority queues, for which there should be standard classes as well as basic system interfaces.

When we have to use a **special data structure**, which is not standardly implemented in .NET Framework, we have two options:

- **First option**: we **implement the data structure ourselves**. This gives us flexibility, because the implementation will completely meet our needs, but it takes a lot of time and it has a great chance of making mistakes.

For example, if one has to qualitatively implement a balanced tree, this may take an experienced software developer several days (along with the tests). If the same is implemented by inexperienced software developer it will take a lot more time and most probably there will be errors in the implementation.

- **Second option** (generally preferable): **find an external library**, which has a full implementation of the needed functionality. This approach has an advantage of saving us time and troubles, because in most cases the external libraries of data structures are well-tested.

They have been used for years by thousands of software developers and this makes them mature and reliable.

**Power Collections for .NET**

**Info:**

One of the most popular and richest libraries with efficient implementations of the fundamental data structures for C# and .NET software developers is the open-source project **"Wintellect’s Power Collections for .NET"**:

Link: <http://powercollections.codeplex.com>.

It provides free, reliable, efficient, fast and handy implementations of the following commonly used **data structures**, which are missing or partly-implemented in .NET framework:

- **Set<T>** – **set** of elements, **implemented with a hash-table**. It efficiently implements the basic operations over sets: adding, deleting and searching an element as well as union, intersection, difference between sets and many more. By functionality and way of work the class looks like the standard class **HashSet<T>** in .NET Framework.

- **Bag<T>** – **multi-set of elements** (set with duplicates), implemented with a **hash-table**. It efficiently implements all basic operations over multi-sets.

- **OrderedSet<T>** – **ordered set of elements** (without duplicates), implemented with a **balanced search tree**. It efficiently implements all basic operations over sets and when traversing through its elements it returns them in ascending order (according to the used comparer). It allows a fast extraction of subsets of values in a given interval.

- **OrderedBag<T>** – **ordered multi-set** of elements, implemented with a **balanced search tree**. It efficiently implements all basic operations over multi-sets and when going through all its elements it returns them in ascending order (according to the used comparer). It allows a quick extraction of subsets of values in a given interval.

- **MultiDictionary<K,T>** – it is a **hash-table allowing key duplicates**. For every key there is a collection of values stored, not one single value.

- **OrderedDictionary<K,T>** – it represents a **dictionary, implemented with a balanced search tree**. It allows a fast search by **key** and whengoing through its elements it returns them in ascending order. Itenables us to quickly extract the values from a given key range. Byfunctionality and way of work the class looks like the standard class **SortedDictionary<K,T>** in .NET Framework.

- **Deque<T>** – represents efficient implementation of a queue with two ends (**double ended queue**), which practically combines the data structures stack and queue. It allows efficient addition, extraction and deletion of elements in both ends.

- **BagList<T>** – **list of elements, accessed through index**, which allows a **quick insertion and deletion** of an element from a particular position. The operations index accessing, adding, inserting at position and removing an element from position have a complexity O(log N). The implementtation is with a balanced tree. The structure is a good alternative of **List<T>**, in which the insertion and removal of element at a particular position takes linear time because of the need of the replacement of linear number of elements to the left or right.

**C5 Collections for .NET**

**Info:**

Another very powerful library of data structures and collection classes is “**The C5 Generic Collection Library for C# and CLI**” (www.itu.dk/research/c5/).

It provides standard interfaces and collection classes like **lists**, **sets**, **bags**, **multi-sets**, **balanced trees** and **hash tables**, as well as **non-traditional** **data structures** like “hashed linked list”, “wrapped arrays” and “interval heaps”. It also describes a set of collection-related **algorithms** and **patterns**, such as “read-only access”, “random selection”, “removing duplicates”, etc.

The library comes with solid documentation (a book of 250 pages). The C5 collections and the book about them are the ultimate resource for data structure developers.

**Object-Oriented**

**Programming Principles**

**Info:**

**Classes** are a description (**model**) of real objects and events referred to as entities.

Classes possess **characteristics** – in programming they are referred to as **properties**.

Classes also expose **behavior** known in programming as **methods**.

Methods and properties can be **visible** only within the scope of the class, which declared them and their descendants (**private** / **protected**), or visible to all other classes (**public**).

**Objects** are **instances** of classes.

**Object-Oriented Programming (OOP)**

**Info:**

Object-oriented programming is the successor of procedural (structural) programming. **Procedural programming** describes programs as groups of reusable code units (procedures) which define input and output parameters.

Procedural programs consist of **procedures**, which invoke each other. The **problem** with procedural programming is that **code reusability is hard** and limited – only procedures can be reused and it is hard to make themgeneric and flexible.

There is no easy way to work with abstract datastructures with different implementations.

**The main advantages and goals of OOP** are to make complex software faster to develop and easier to maintain. OOP enables the easy reuse of code by applying simple and widely accepted rules (principles).

**Fundamental Principles of OOP**

**Info:**

In order for a programming language to be **object-oriented**, it has to enable working with **classes** and **objects** as well as the implementation and use of the fundamental object-oriented principles and concepts: **inheritance**, **abstraction**, **encapsulation** and **polymorphism**. Let’s summarize each of these **fundamental principles of OOP**:

- **Encapsulation** 🡺 **hide unnecessary details** in our classes and provide aclear and simple interface for working with them.

- **Inheritance** 🡺 **class hierarchies** improve code readability and enable the reuse of functionality.

- **Abstraction** 🡺 **work through abstractions**: to deal with objects considering their important characteristics and ignore all other details.

- **Polymorphism 🡺** how to work in the same manner with different objects,which define a specific implementation of some **abstract behavior**.

Some OOP theorists also put the concept of **exception handling** as additional **fifth fundamental principle of OOP**. We shall not get into a detailed dispute about whether or not exceptions are part of OOP and rather will note that **exceptions are supported in all modern object-oriented** **languages** and are the primary mechanism of handling errors and unusual situations in object-oriented programming. **Exceptions always come** **together with OOP.**

**1. Inheritance**

**Info:**

**Inheritance** is a fundamental principle of object-oriented programming. It allows a class to "inherit" (behavior or characteristics) of another, more general class. For example, a lion belongs to the biological family of cats (Felidae). All cats that have four paws, are predators and hunt their prey.

This functionality can be coded once in the **Felidae** class and all its predators can reuse it – **Tiger**, **Puma**, **Bobcat**, etc.

Inheritance is described as **is-kind-of** **relationship**, e.g. **Tiger** is kind of **Animal**.

**How Does Inheritance Work in .NET?**

**Info:**

Inheritance in .NET is defined with a special construct in the class declaration. In .NET and other modern programming languages, a class can inherit from a single class only **(single inheritance)**, unlike C++ which supports inheriting from multiple classes **(multiple inheritance)**.

This limitation is necessitated by the difficulty in deciding which method to use when there are duplicate methods across classes (in C++, this problem is solved in a very complicated manner).

In .NET, classes can inherit multiple interfaces. The class from which we inherit is referred to as **parent class** or **base class** **/ super class**.

**The "base" Keyword**

**Info:**

In the above example, we used the **keyword base** in the constructor of the class **Lion**. The keyword indicates that **the base class must be used and allows access to its methods, constructors and member variables**.

Using**base()**, we can call the constructor of the base class. Using **base.Method(…)** we can invoke a method of the base class, pass parameters to it and use its results.

Using **base.field** we can get the value of a member variable from the base class or assign a different one to it.

In .NET, methods inherited from the base class and declared as **virtual** can be **overridden**. This means **changing their implementation**; the original source code from the base class is ignored and new code takes its place.

We can invoke non-overridden methods from the base class without using the keyword **base**. Using the keyword is required only if we have an overridden method or variable with the same name in the inheriting class.

**The keyword base can be used explicitly for clarity.**

**base.method(…) calls a method, which is necessarily from the base class. Such source code is easier to read, because we know where to look for the method in question.**

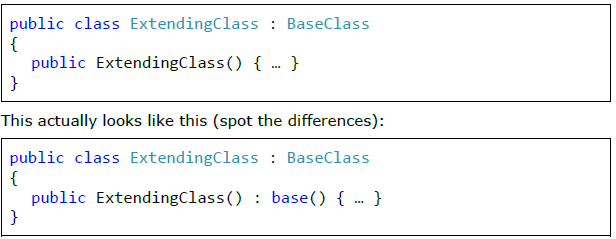
**Bear in mind that using the keyword this is not the same. It can mean accessing a method from the current, as well as the base class.**

**Constructors with Inheritance**

**Info:**

When inheriting a class, our constructors must call the base class constructor, so that it can **initialize its member variables**.

If we do not do this explicitly, **the compiler will place a call to the parameterless base class constructor**, "**:base()**", at the beginning of all our inheriting class' constructors. Here is an example:



**If the base class has no default constructor (one without parameters) or that constructor is hidden, our constructors need to explicitly call one of the other base class constructors. The omission of such a call will result in a compile-time error!**

**If a class has private constructors only, then it cannot be inherited!**

**If a class has private constructors only, then this could indicate many other things. For example, no-one (other than that class itself) can create instances of such a class. Actually, that’s how one of the most popular design patterns (Singleton) is implemented.**

**Calling the constructor of a base class happens outside the body of the constructor.**

**The idea is that the fields of the base class should be initialized before we start initializing fields of the inheriting class, because they might depend on a base class field.**

**Access Modifiers of Class Members and Inheritance**

**Info:**

Let’s review: in the "Defining Classes" chapter, we examined the **basic access modifiers**.

Regarding members of a class (**methods**, **properties** and **member** **variables**) we examined the modifiers **public**, **private** and **internal**.

Actually, there are two other modifiers: **protected** and **protected** **internal**.

**This is what they mean:**

- **protected** defines class members which are not visible to users of the class (those who initialize and use it), but are visible to all inheriting classes (descendants).

- **protected internal** defines class members which are both **internal**, i.e. visible within the entire assembly, and **protected**, i.e. not visible outside the assembly, but visible to classes who inherit it (even outside the assembly).

**When a base class is inherited:**

- All of its **public**, **protected** and **protected internal** members (methods, properties, etc.) are **visible** to the inheriting class.

- All of its **private** methods, properties and member-variables are **not visible** to the inheriting class.

- All of its **internal** members are visible to the inheriting class, only if the base class and the inheriting class are **in the same assembly** (the same Visual Studio project).

**The System.Object Class**

**Info:**

Object-oriented programming practically became popular with **C++**. In this language, it often becomes necessary to code classes, which must work with objects of any type. C++ solves this problem in a way that is not considered strictly object-oriented (by using **void** pointers).

The architects of .NET take a different approach. They create a class, which **all other classes inherit** (directly or indirectly).

**All objects can be perceived as instances of this class.** It is convenient that this class contains important methods and their default implementation.

This class is called **Object** (which is the same as **object** and **System.Object**).

**In .NET every class, which does not inherit a class explicitly, inherits the system class System.Object by default. The compiler takes care of that.**

Every class, which inherits from another class indirectly, inherits **Object** from it. This way every class inherits explicitly or implicitly from **Object** and contains all of its fields and methods.

Because of this property, **every class instance can be cast to Object**. A typical example of the advantages of implicit inheritance is its use with data structures, which we saw in the chapters on data structures.

Untyped list structures (like **System.Collections.ArrayList**) can hold all kinds of objects, because they treat them as instances of the class **Object**.

**The generic types (generics) have been provided specifically for working with collections and objects of different types.**

**They allow creating typified classes, e.g. a collection which works only with objects of type (ex. Lion).**

**.NET, Standard Libraries and Object**

**Info:**

In .NET, there are a lot of predefined classes (we already covered a lot of them in the chapters on collections, text files and strings).

These classes are part of the .NET framework; they are available wherever .NET is supported.

These classes are referred to as **Common Type System (CTS)**.

.NET is one of the first frameworks, which provide such an extensive set of predefined classes. A lot of them work with **Object** so that they can be used in as many situations as possible.

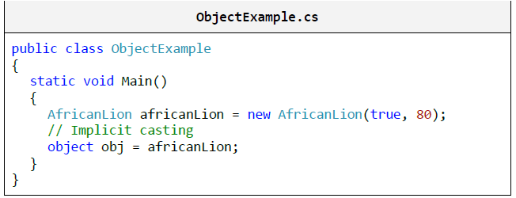
.NET also provides a lot of libraries, which can be referenced additionally, and it stands to reason that they are called class libraries or external libraries.

**Now it is the time to mention that the keywords string and object are simply compiler tricks and are substituted with System.String and System.Object during compilation.**

**The Base Type Object Upcasting and Downcasting – Example**

**Info:**

Let’s take a closer look at the **Object class** using an example:



In this example, we cast an **AfricanLion** to **Object**. This operation is called **upcasting** and is permitted because **AfricanLion** is an indirect child of the **Object** class.

In this example, we **cast an Object to AfricanLion**. This

operation is called **downcasting** and is permitted only if we indicate

the type we want to cast to, because **Object** is a parent class of **AfricanLion** and it is not clear if the variable **obj** is of type **AfricanLion**. If it is not, an **InvalidCastException** will be thrown.



**The Object.ToString() Method**

**Info:**

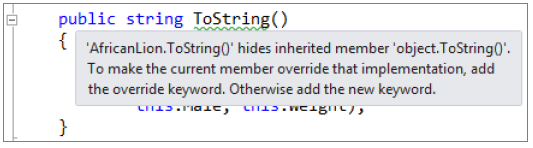
One of the most commonly used methods, originating from the class **Object** is **ToString()**. It returns a **textual representation of an object**.

Every object includes this method and therefore has a textual representation. This method is used when we print the object using **Console.WriteLine(…)**.

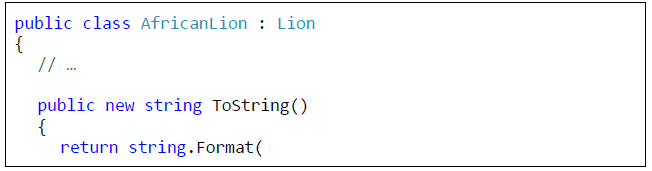
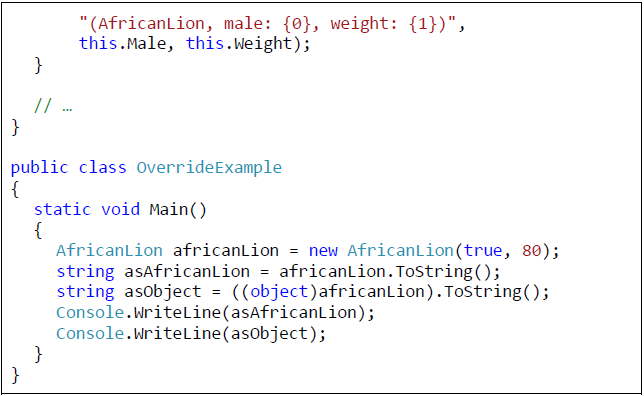
**Virtual Methods: the "override" and "new" Keywords**

**Info:**

We need to explicitly instruct the compiler that we want our method to **override** another. In order to do this, we use the **override keyword**. Notice what happens if we remove it:

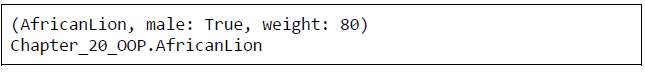


Let’s experiment and use the **keyword new** instead of **override**:



This is the

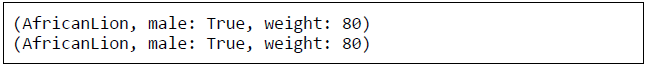
result:



We notice that the implementation of **Object.ToString()** is invoked when we upcast **AfricanLion** to **object**. In other words, when we use the keyword **new**, we create a new method, which hides the old one.

The old method can then only be called with an upcast.

What would happen, if we reverted to using the keyword **override** in the previous example?



Surprising, isn’t it? It turns out that when we override a method, we cannot access the old implementation even if we use upcasting.

This is because there are no longer two **ToString()** methods, but rather only the one we overrode.

A method, which can be overridden, is called **virtual**. In .NET, methods are not **virtual** by default. If we want a method to be overridable, we can do so by including the keyword **virtual** in the declaration of the method**!**

The explicit instructions to the compiler that we want to override a method (by using **override**), is a protection against mistakes.

If there’s a typo in the method’s name or the types of its parameters, the compiler will inform us immediately of this mistake. It will know something is not right when it cannot find a method with the same signature in any of the base classes.

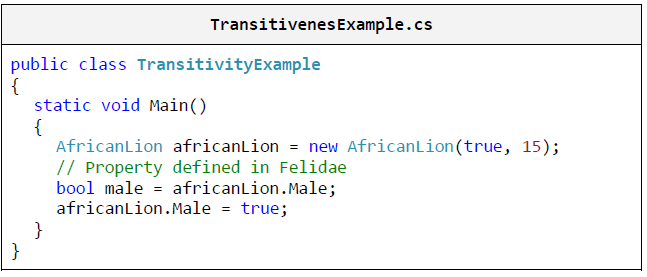
**Transitive Properties of Inheritance**

**Info:**

In mathematics, **transitivity** indicates transferability of relationships. Let’s take the indicator "larger than" (>) as an example. If A>B and B>C, we can **conclude** that A>C. This means that the relation "larger than" (>) is transitive, because we can unequivocally determine whether A is larger or smaller than C and vice versa.

If the class **Lion** inherits the class **Felidae** and the class **AfricanLion** inherits **Lion**, then this implies that **AfricanLion** inherits **Felidae**. Therefore, **AfricanLion** also has the property **Male**, which is defined in **Felidae**.

Thisuseful property allows a particular functionality to be defined in the mostappropriate class.



It is because of the transitive property of inheritance that we can be sure that all classes include the method **ToString()** and all other methods of **Object** regardless of which class they inherit.

**Inheritance Hierarchy**

**Info:**

If we try to describe all big cats, then, sooner or later, we will end up with a relatively large group of classes, which inherit one another. All these classes, combined with the base classes, form a **hierarchy** of big cat classes.

The easiest way to describe such hierarchies is by using **class diagrams**. Let’s take a look at what a "class-diagram" is.

**Class Diagrams**

**Info:**

A **Class Diagram** is one of several types of diagrams defined in UML. **UML (Unified Modeling Language)** is a notation for visualizing different processes and objects related to software development.

**Class diagrams** are used to describe visually class hierarchies, inheritance and the structure of the classes themselves.

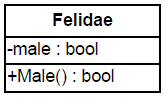
Two pieces of UML terminology are:

**-** the first one is **generalization**. Generalization is a term signifying the **inheritance of a class** or the **implementation of an interface.**

**-** the other term is **association**. An association, would be, e.g. "The Lion has paws", where **Paw** is another class. Association is **has-a relationship**.

**Generalization and association are the two main ways to reuse code.**

**A Class Based on a Class Diagram – Example**

**Info:**

This is what a sample class diagram looks like:

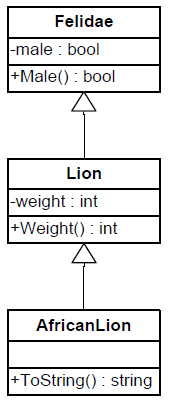
The class is represented as a **rectangle**, divided in

3 boxes one under another. The **name** of the class

is at the top. Next, there are the **attributes** (UML term)

of the class (in .NET they are called member variables and properties). At the very bottom are the **operations** (UML term) or methods (in .NET jargon). The plus/minus signs indicate whether an attribute / operation is visible (**+** means **public**) or not visible (**-** means **private**). **Protected** members are marked with **#**.

**Class Diagram – Example of Generalization**

**Info:**

Here is a class diagram that visually illustrates

generalization (**Felidae** inherited by **Lion** inherited

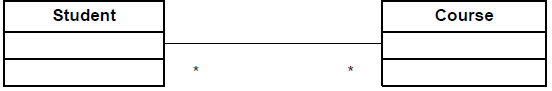
by **AfricanLion**):

**Associations**

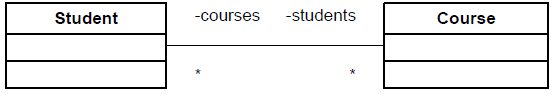
**Info:**

Associations denote **connections between classes**. They model mutual relations. They can define **multiplicity** (1 to 1, 1 to many, many to 1, 1 to 2, …, and many to many).

**-** A **many-to-many** association is depicted in the following way:

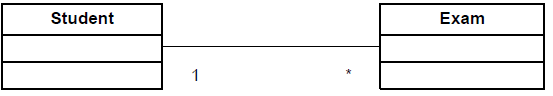


**-** A **many-to-many association by attribute** is depicted in the following way:

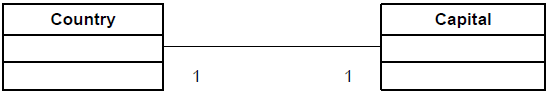


In this case, there are connecting attributes, which indicate the variables holding the connection between classes.

**-** A **one-to-many** association is depicted like this:



**-** A **one-to-one** association is depicted like this:

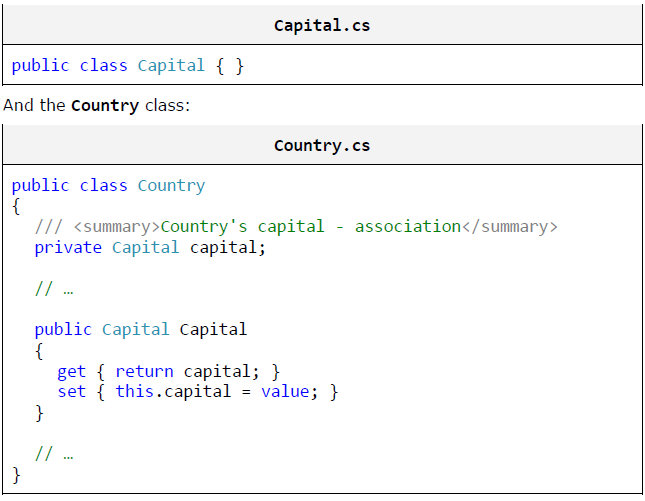


**From Diagrams to Classes**

**Info:**

Class diagrams are most often used for creating classes. Diagrams facilitate and speed up the **design of classes in a software project**.

We can create classes directly following the diagram above. Here is the **Capital** class:



**Aggregation**

**Info:**

Aggregation is a special type of association. It models the **relationship of kind "whole / part"**. We refer to the parent class as an **aggregate.**

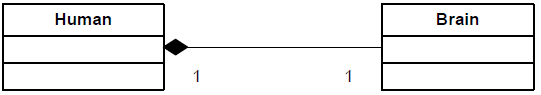
The aggregated classes are called **components**. There is an empty rhombus at one end of the aggregation:



**Composition**

**Info:**

A filled rhombus represents composition. Composition is an aggregation where the **components cannot exist without the aggregate**:



**2. Abstraction**

**Info:**

**Abstraction means working with something we know how to use without knowing how it works internally**.

A good example is a television set. We don’t need to know the inner workings of a TV, in order to use it. All we need is a remote control with a small set of buttons (the interface of the remote) and we will be able to watch TV.

**Abstraction** is something we do every day. This is an action, which obscures all details of a certain object that do not concern us and only uses the details, which are relevant to the problem we are solving.

For example, in hardware configurations, there is an **abstraction called "data storage device"** which can be a **hard disk**, **USB memory stick** or **CD-ROM drive**. Each of these works in a different way internally but, from the point of view of the operating system and its applications, it is used in the same way – it **stores files and folders**.

In Windows we have Windows Explorer and it can work with all devices in the same way, regardless of whether a device is a hard drive or a USB stick. It works with the **abstraction** "storage device" and is not involved with how data is read or written. The drivers of the particular device take care of that. **They are implementations of the interface "data storage device"**.

Abstraction is one of the **most important concepts** in programming and OOP.

It allows us to write **code, which works with abstract data** **structures** (like dictionaries, lists, arrays and others). We can work with an abstract data type by using its interface without concerning ourselves with its implementation.

For instance, we can save to a file all elements from a list

without bothering if it is implemented with an array, a linked list, etc.

The code remains unchanged, when we work with other data types. We can even write new data types (we will discuss this later) and make them work with our program without changing it.

Abstraction allows us to do something very important – **define an interface for our applications**, i.e. to define a**ll tasks the program is capable to execute** and their respective input and output data.

That way we can make acouple of small programs, each handling a smaller task. When we combinethis with the ability to work with **abstract data**, we achieve great flexibility inintegrating these small programs and much more opportunities for codereuse.

These small subprograms are referred to as **components**. Thisapproach for writing programs is widely adopted since it allows us to reusenot only objects, but entire subprograms as well.

**Interfaces**

**Info:**

In the C# language the **interface** is a definition of a **role** (a group of abstract actions). It defines what sort of behavior a certain object must exhibit, without specifying how this behavior should be implemented.

Interfaces are also known as **contracts** or **specifications** of behavior.

An object can have **multiple roles** (or **implement multiple interfaces** / contracts) and its users can utilize it from different points of view.

For example, an object of type **Person** can have the roles of **Soldier** (with behavior "shoot your enemy"), **Husband** (with behavior "love your wife") and **Taxpayer** (with behavior "pay your taxes").

However, every person implements its behavior in a different way; **John** pays his taxes on time, **George** pays them overdue and **Peter** doesn’t pay them at all.

Some may ask why the base class of all objects (the class **Object**) is not an interface. The reason is because in such case, every class would have to implement a small, but very important group of methods and this would take an unnecessary amount of time.

It turns out that not all classes need a specific implementation of **Object.GetHashCode()**, **Object.Equals(…)** and **Object.ToString()**, i.e. the default implementation suffices in most cases. It’s not necessary to override any of the methods in the **Object** class, but if the situation calls for it we can.

**Interfaces – Key Concepts**

**Info:**

An **interface can only declare methods and constants**.

A **method signature** is the combination of a method’s **name** and a description of its **parameters** (type and order).

In a class / interface all methods have to have different signatures and should not be identical with signatures of inherited methods.

A **method declaration** is the combination of a method’s **return type and its signature**. The return type only specifies what the method returns.

**A method is identified by its signature. The return type is not a part of it. If two methods' only difference is the return type (as in the case when a class inherits another), then it cannot be unequivocally decided which method must be executed.**

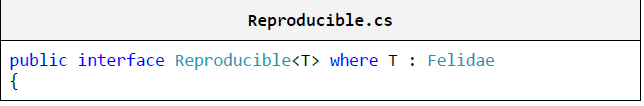
A **class / method implementation** is the source code of a class / method. Usually it is between curly brackets: "**{**" and "**}**". Regarding methods, this is also referred to as the **method body**.

**Interfaces – Example**

**Info:**

An interface in .NET is defined with the **keyword interface**. An interface can contain only method declarations and constants.

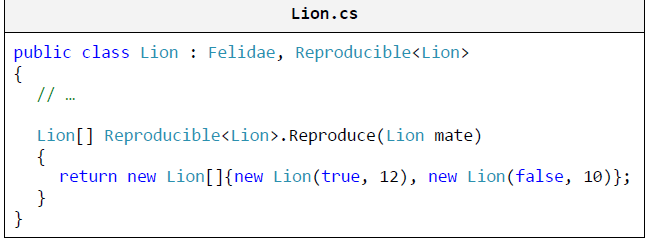
Here is an example of an interface:





The interface we wrote has a method of type **T** (**T** must inherit **Felidae**) which returns an array of **T**.

And this is how the class **Lion**, which **implements the interface Reproducible** looks like:



The name of the interface is coded in the declaration of the class (on the first row) and specifies the generic class.

We can indicate which method from a specific interface we implement by typing its name explicitly:



In an interface, methods are only declared; the implementation is coded in the class implementing the interface, i.e. – **Lion**.

**The class that implements a certain interface must implement all methods in it.**

**The only exception is when the class is abstract. Then it can implement none, some or all of the methods. All remaining methods have to be implemented in some of the inheriting classes.**

**Abstraction and Interfaces**

**Info:**

The best way to achieve abstraction is by **working though interfaces**. A component works with interfaces which another implements. That way, a change in the second component will not affect the first one as long as the new component implements the old interface.

The interface is also called a **contract**. Every component upholds a certain contract (the signature of certain methods). That way, two components upholding a contract can communicate with each other without knowing how their counterpart works.

Some important interfaces from the Common Type System (CTS) are the **list** and **collection** interfaces: **System.Collections.Generic.IList<T>** and **System.Collections.Generic.ICollection<T>**.

All of the standard .NETcollection classes implement these interfaces and the various componentspass different implementations (arrays, linked lists, hash tables, etc.) to oneanother using a common interface.

**Collections are an excellent example of an object-oriented library with classes and interfaces that actively use all core principles of OOP: abstraction, inheritance, encapsulation and polymorphism.**

**When Should We Use Abstraction and Interfaces?**

**Info:**

The answer to this question is: **always when we want to achieve abstraction of data or actions**, whose implementation can change later on.

Code, which communicates with another piece of code through interfaces, is much more resilient to changes than code written using specific classes.

**Working through interfaces is common and a highly recommended practice** – one of the basic rules for writing high-quality code.

**When Should We Write Interfaces?**

**Info:**

It is always a good idea to use interfaces **when functionality is exposed to another component**.

In the interface we include only the functionality (inthe form of a declaration) that others need to see.

Internally, a program / component can use interfaces for **defining roles**. That way, an object can be used by different classes through different roles.

**3.** **Encapsulation**

**Info:**

**Encapsulation** is one of the main concepts in OOP. It is also called "**information hiding**".

An object has to provide its users only with the essential information for manipulation, without the internal details.

A **Secretary** using a **Laptop** only knows about its screen, keyboard and mouse. Everything else is hidden internally under the cover. She **does not know** **about the inner workings** of **Laptop**, because she doesn’t need to, and if she does, she might make a mess.

Therefore parts of the properties and methods remain hidden to her. The person writing the class has to decide what should be **hidden** and what not.

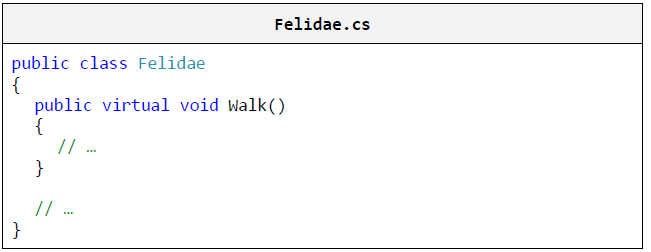
**When we program, we must define as private every method or field which other classes should not be able to access.**

**Encapsulation – Examples**

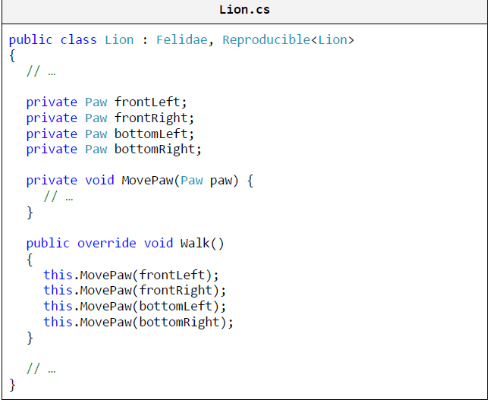
**Info:**

The example below shows how to **hide methods** that the class’ user doesn’t have to be familiar with and are only used internally by the author of the class.

First, we define an **abstract class Felidae**, which defines the public operations of cats (regardless of the cat’s type):



This is how the class **Lion** looks like:



The public method **Walk()** calls some other

**private** method 4 times. That way the base

class is short – it consists of a single

method. The implementation, however,

calls another of its methods, which is

hidden from the users of the class. That

way, **Lion doesn’t publicly disclose**

**information about its inner workings**

(it encapsulates certain behavior).

At a later stage, this makes it possible to

change its implementation without any of

the other classes finding out and requiring

changes.

**4. Polymorphism**

**Info:**

The next fundamental principle of Object-Oriented Programming is "Polymorphism". **Polymorphism allows treating objects of a derived** **class as objects of its base class**.

For example, big cats (base class) catch their prey (a method) in different ways. A Lion (derived class) sneaks on it, while a Cheetah (another derived class) simply outruns it.

Polymorphism allows us to treat a cat of random size just like a big cat and command it "catch your prey", regardless of its exact size.

Polymorphism can bear strong resemblance to abstraction, but it is mostly related to **overriding methods in derived classes**, in order **to change** **their original behavior** inherited from the base class.

**Abstraction** is associated with **creating an interface of a component or functionality** (**defining a role**).

**Abstract Classes**

**Info:**

What happens if we want to specify that the class **Felidae** is **incomplete** and only its successors can have instances? This is accomplished by putting **the** **keyword abstract** before the name of the class and indicates that the class is **not ready to be instantiated**.

We refer to such classes as **abstract** **classes**. And how do we indicate which exact part of the class is incomplete?

Once again, this is accomplished by putting the keyword **abstract** before the name of the method to be implemented. This method is called an **abstract** **method** and cannot have an implementation, but a declaration only.

Each class with **at least one abstract method** must be abstract. Makes sense, right? **However, the opposite is not true.** It is possible to define a class as an abstract one, even when there are no abstract methods in it.

Abstract classes are something **in the middle between classes and interfaces**. They can define **ordinary methods** and **abstract methods**.

Ordinary methods have an implementation, whereas abstract methods are **empty** (without an implementation) **and remain to be implemented later by the derived classes.**

**Purely Abstract Classes**

**Info:**

Abstract classes, as well as interfaces, **cannot be instantiated**. If we try to create an instance of an abstract class, we are going to get an error during compilation.

**Sometimes a class can be declared abstract, even if it has no abstract methods, in order to simply prohibit using it directly without creating an instance of a successor!**

A **pure abstract class** is an abstract class, which has no implemented methods and no member variables. **It is very similar to an interface.**

The fundamental difference is that **a class can implement many interfaces and inherit only one class** (even if that class is abstract).

Initially, interfaces were not necessary in the presence of "multiple inheritance". They had to be conceived as a means to supersede it in specifying the numerous roles of an object.

**Virtual Methods**

**Info:**

A method, which can be overridden in a derived class, is called a **virtual method**.

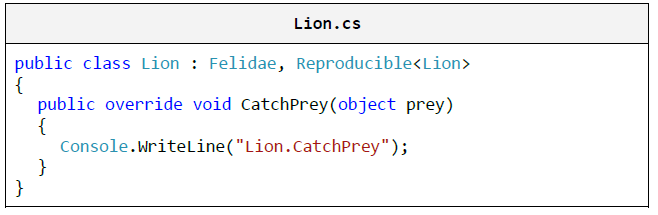
Methods in .NET by default aren’t virtual. If we want to make a method virtual, we mark it with **the keyword virtual**. Then the derived class can declare and define a method with the same signature.

Virtual methods are important for **method overriding**, which lies at the heart of polymorphism.

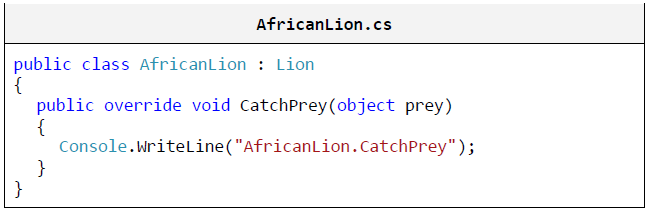
**Virtual Methods – Example**

**Info:**

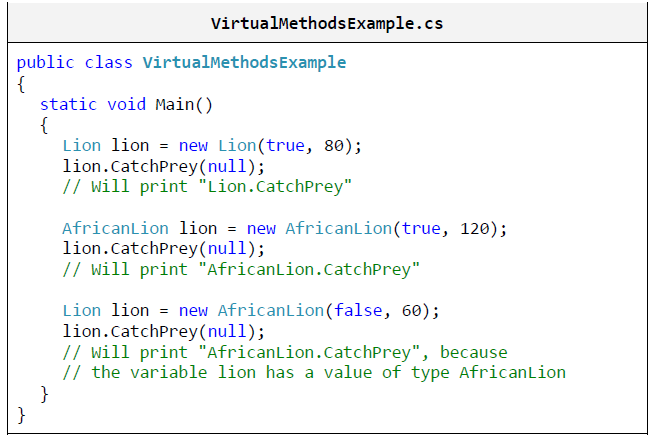
We have a class inheriting another and the two classes share a common method. Both versions of the method write on the console. Here is how the **Lion** class looks like:



Here is how the **AfricanLion** class looks like:



We make three attempts to create instances and call the method **CatchPrey**.



In the last attempt, you can clearly see how, in fact, **the overwritten method is called** and not the base method. This happens, because it isvalidated what the actual class behind the variable is and whether itimplements (overwrites) that method.

**Rewriting of methods** is also called **overriding of virtual methods**.

**Virtual methods** as well as abstract methods can be overridden. Abstract methods are actually virtual methods without a specific implementation. All methods defined in an interface are abstract and therefore virtual, although this is not explicitly defined.

**The Difference between Virtual and Non-Virtual Methods**

**Info:**

The difference between the virtual and non-virtual methods is:

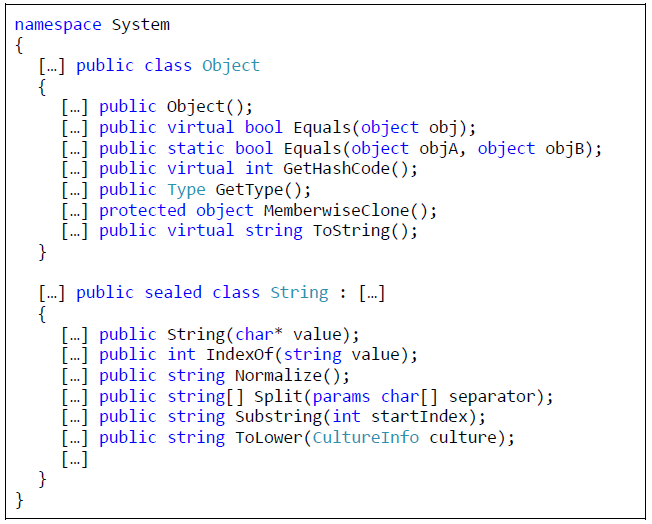
**Virtual methods** are used when we expect from derived classes to change / complement / **alter some of the inherited functionality**.

For example, the method **Object.ToString()** allows derived classes to change / replace its implementation in any way they want. Then, even if we work with an object not directly, but rather by upcasting it to **Object**, we use the overwritten implementation of the virtual methods.

Virtual methods are a key characteristic of objects when we talk about abstraction and working with abstract types.

**Sealing of methods** is done when we rely on a piece of functionality and we don’t want it to be altered. We already know that methods are **sealed** by default. But if we want a base class’ virtual method to become sealed in a derived class, we use **override sealed**.

The **string class has no virtual methods**. In fact, inheriting **string** is entirely **forbidden for inheritance** through the keyword **sealed** in its declaration.

Here are parts of the declarations of **string** and **object** classes (the ellipses in square brackets indicate omitted, irrelevant code):

**When Should We Use Polymorphism?**

**Info:**

The answer to this question is simple: whenever we want **to enable changing a method’s implementation in a derived class**.

It’s a good ruleto work with the most basic class possible or directly with an interface. Thatway, changes in used classes reflect to a much lesser extent on classeswritten by us. The less a program knows about its surrounding classes, thefewer changes (if any) it would have to undergo.

**Cohesion and Coupling**

**Info:**

The terms **cohesion and coupling** are inseparable from OOP. They complement and explain further some of the principles we have described so far. Let’s get familiar with them.

**Cohesion**

**Info:**

The concept of **cohesion** shows to what degree a program’s or a component’s various tasks and responsibilities are **related to one another**, i.e. how much a program is focused on solving a single problem. Cohesion is divided into **strong cohesion** and **weak cohesion**.

**Strong Cohesion**

**Info:**

Strong cohesion indicates that the responsibilities and tasks of a piece of code (a method, class, component or a program) are **related to one another** and intended to **solve a common problem**.

This is something we must always aim for. Strong cohesion is a typical characteristic of high-quality software.

**Strong Cohesion in a Class**

**Info:**

Strong cohesion in a class indicates that the class **defines only one entity**. An entity can have many roles (Peter is a soldier, husband and a taxpayer). Each of these roles is defined in the same class.

Strong cohesion indicates that the class solves only one task, one problem, and not many at the same time.

A class, which does **many things at the same time**, is difficult to understand and maintain. Consider a class, which implements a hash table, provides functions for printing, sending an e-mail and working with trigonometric functions all at once. How do we name such a class? If we find it difficult to answer this question,

It means that we have failed to achieve **strong cohesion** and have to separate the class into several smaller classes, each solving a single task.

**Strong Cohesion in a Class – Example**

**Info:**

As an example of strong cohesion we can point out the **System.Math** class. It performs **a single task**: it provides mathematical calculations and constants:

- **Sin()**, **Cos()**, **Asin()**

- **Sqrt()**, **Pow()**, **Exp()**

- **Math.PI**, **Math.E**

**Strong Cohesion in a Method**

**Info:**

**A method is well written when it performs only one task and performs it well**. A method, which does a lot of work related to different things, has **bad cohesion**. It has to be **broken down into simpler methods**, eachsolving only one task.

Once again, the question is posed what name shouldwe give to a method, which finds prime numbers, draws 3D graphics on thescreen, communicates with the network and prints records extracted from adata base?

Such a method has **bad cohesion** and has to be logicallyseparated into several methods.

**Weak Cohesion**

**Info:**

Weak cohesion is observed along with **methods, which perform several unrelated tasks**.

Such methods take several different groups of parameters,

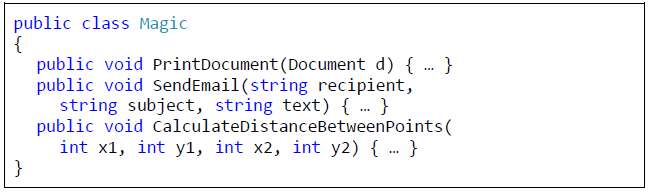
in order to perform different tasks. Sometimes, this requires logically

unrelated data to be unified for the sake of such methods.

**Weak cohesion is harmful and must be avoided!**

**Weak Cohesion – Example**

Here is a sample class with weak cohesion:



**Best Practices with Cohesion**

**Info:**

**Strong cohesion** is quite logically the "good" way of writing code. The concept is associated with simpler and clearer source code – **code that is easier to maintain and reuse** (because of the fewer tasks it has to perform).

Contrarily, with **weak cohesion** each change is a ticking time bomb, because it could affect other functionality. Sometimes a logical task is spread out to several different modules and thus changing it is more labor intensive. Code reuse is also difficult, because a component does several unrelated tasks and to reuse it the exact same conditions must be met which is hard to achieve.

**Coupling**

**Info:**

Coupling mostly describes the extent to which components / classes **depend on one another**. It is broken down into **loose coupling** and **tight coupling**.

Loose coupling usually correlates with strong cohesion and vice versa.

**Loose Coupling**

**Info:**

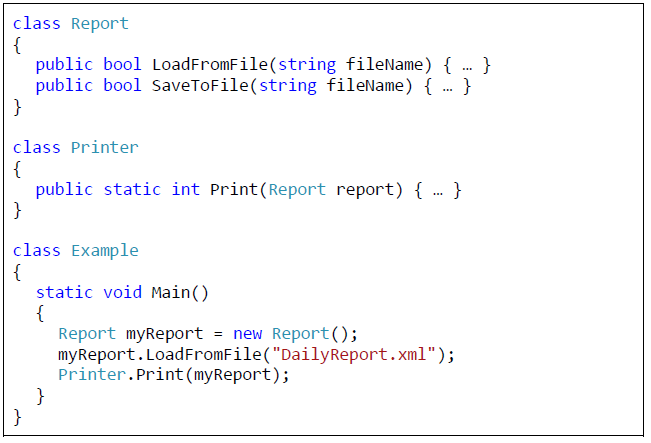
Loose coupling is defined by a piece of code’s (program / class / component) communication with other code through **clearly defined interfaces** (contracts).

A change in the implementation of a loosely coupled component doesn’t reflect on the others it communicates with. When you write source code, you **must not rely on inner characteristics** of components (specific behavior that is not described by interfaces).

The contract has to be maximally simplified and define only the required behavior for this component’s work by hiding all unnecessary details.

**Loose coupling is a code characteristic you should aim for. It is one of the characteristics of high-quality programming code!**

**Loose Coupling – Example**

Here is an **example of loose coupling** between classes and methods:

In this example, **none of the methods depend on the others**. The methods rely only on some of the parameters, which are passed to them.

Should we need one of the methods in a next project, we could easily take it out and reuse it.

**Tight Coupling**

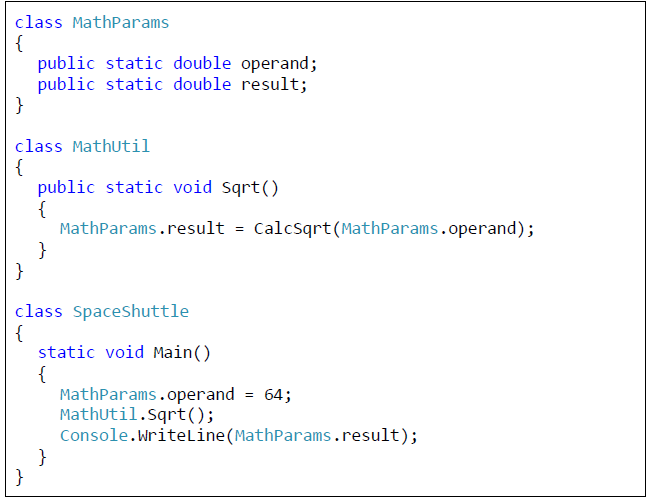
**Info:**

We achieve tight coupling when there are many input parameters and output parameters; when we use undocumented (in the contract) characteristics of another component (for example, a dependency on static fields in another class); and when we use many of the so called control parameters that indicate behavior with actual data.

**Tight coupling** between two or more methods, classes or components means that **they cannot work** **independently of one another** and that a change in one of them will also affect the rest.

This leads to difficult to read code and big problems with its maintenance.

**Tight Coupling – Example**

Here is an **example of tight coupling** between classes and methods:

Such code is **difficult to understand and maintain**, and the likelihood of mistakes when using it is great.

Think about what happens if another method, which calls **Sqrt()**, passes its parameters through the same static variables

**operand** and **result**.

If we have to use the same functionality for deriving square root in a subsequent project, we will not be able to simply copy the method **Sqrt()**, but rather we will have to copy the classes **MathParams** and **MathUtil** together with all of their methods.

This makes the code difficult to reuse. In fact, the above code is an **example of bad code** according to all rules of Procedural and Object-Oriented Programming and if you think twice, you will certainly identify at least several more disregarded recommendations from those we have given you so far.

**Best Practices with Coupling**

**Info:**

The most common and advisable way of invoking a well written module’s functionality is through interfaces. That way, the functionality can be substituted without clients of the code requiring changes.

The jargon expression for this is "**programming against interfaces**".

Most commonly, an interface describes a "**contract**" observed by this module. It is good practice not to rely on anything else other than what’s described by this contract.

The use of inner classes, which are not part of the public interface of a module, is not recommended because their implementation can be substituted without substituting the contract.

It is good practice that the **methods are made flexible** and ready to work with all components, which observe their interfaces, and not only with definitive.

The latter would mean that these methods expect something specific from the components they can work with. It is also good practice that **all dependencies are clearly described and visible**. Otherwise, the maintenance of such code becomes difficult.

Let’s take, for instance, a **hard disk drive (HDD)**:

It **solves only one task** doesn’t it? The hard disk solves the task of storing data. It does not cool down the computer, does not make sounds, has no computing power and is not used as a keyboard.

It is connected to the computer with two cables only, i.e. it has a **simple interface** for access and is not bound to other peripherals. The hard disk **works separately** and other devices aren’t concerned about how it works exactly.

The CPU commands it to "read" and it reads, then it commands it to "write" and it writes. How exactly it does this remains **hidden inside** it. Different models can work in different ways, but that is their own concern.

You can see that the CPU has strong cohesion, loose coupling, good abstraction and good encapsulation.

This is how you should implement your classes – they must **do only one thing**, **do it well**, **bind them minimally to other classes** (or not link them at all whenever that’s possible), have a clear interface and good abstraction and to hide the details of their internal workings.

**Object-Oriented Modeling (OOM)**

**Info:**

**Object-oriented modeling (OOM**) is a process associated with OOP where all objects related to the problem we are solving are brought out (a model is created).

Only the classes' characteristics, which are important for solving this particular problem, are elicited. The rest are ignored. That way, we create a new reality, a **simplified version of the original** one (its model), such that it allows us to solve the problem or task.

For example, if we model a **ticketing system**, the **important characteristics** of a passenger could be their name, their age, whether they use a discount and whether they are male or female.

A passenger has **many other not important characteristics we aren’t concerned about**, such as the color of their eyes, what shoe size theywear, what books they like or what beer they drink.

By modeling, **a simplified model of reality is created** in order to solve a specific task. In object-oriented modeling, the model is created by means of OOP: via classes, class attributes, class methods, objects, relations between classes, etc. Let’s scrutinize this process.

**When a class becomes large and complicated it has to be broken down into several smaller classes.**

**Use Case Diagrams**

**Info:**

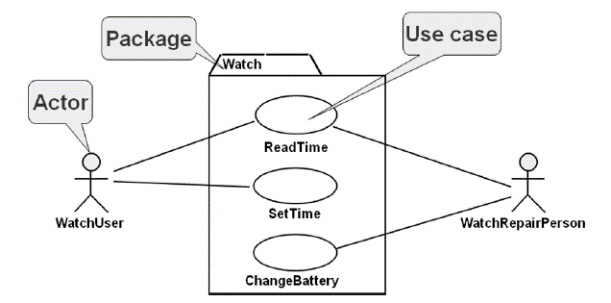
They are used when we elicit the requirements for the description of possible actions. **Actors** represent roles (types of users).

**Use cases** describe interaction between the actors and the system. The use case model is a group of use cases – it provides a complete description of a system’s functionality.

**Use Case Diagrams – Example**

**Info:**

Here is how a **use case diagram** looks like:



**Sequence Diagrams**

**Info:**

Sequence diagrams are used when modeling the requirements of **process specification** and describing use case scenarios more extensively.

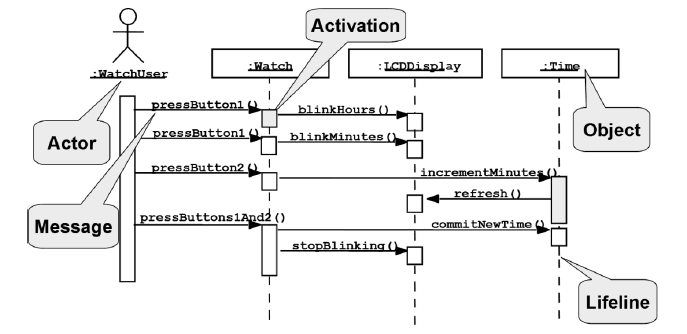
They allow describing additional participants in the processes and the sequence of the actions over the time. They are used in designing the descriptions of system interfaces.

Sequence diagrams describe **what happens over the time**, the interactions over the time, the **dynamic view** over the system, a **sequence of steps**, just like an algorithm.

**Sequence Diagrams – Example**

**Info:**

Here is how a sequence diagram looks like:



**Design Patterns**

**Info:**

**Design patterns** appeared as proven and highly-efficient **solutions to the most common problems of object-oriented modeling**.

**The Singleton Design Pattern**

**Info:**

This is the most popular and most frequently used design pattern. It allows a **class to have only one instance** and defines where it has to be taken from.

Typical examples are classes, which define references to singular entities (a virtual machine, operating system, window manager in a graphical application or a file system) as well as classes of the next pattern (factory).

**The Factory Method Design Pattern**

**Info:**

**Factory method** is another very common design pattern. It is intended for **"producing" objects**.

The instantiation of an object is not performed directly, but rather by the factory method. This allows the factory method to decide which specific instance to create from a family of classes implementing a common interface.

The solution can depend on the environment, a parameter or some system setting.

Factory methods **encapsulate object creation**. This is useful if the creation process is very complicated – if it depends on settings in configuration files or input data by the user.

**High-Quality Programming Code**

**Identifier Naming**

**Info:**

When naming an identifier, it is good to ask yourself these questions:

- What does this class do?

- What is the purpose of this variable?

- What is that method being used for?

- What information does this parameter hold?

**The name of an identifier should describe its purpose. The solution of problem 12 from the exercises should NOT be called Problem12. That is a huge mistake!**

**Use consistent names: use the same words for the same situations, do not use synonyms. Name opposite things symmetrically!**

In .NET there is one more notation for naming interfaces: naming them so that they end in "**able"**: **ICloneable**, **IEnumerable**, **IFormattable**.

These are interfaces that most often augment the basic role of an object. Most interfaces, however, do not follow this notation, such as the **IList** and **ICollection** interfaces.

**Methods should have one purpose only, solving only one task, not multiple tasks at the same time!**

Methods solving multiple tasks (**weak cohesion**) cannot and should not be named properly. They must be **refactored**.

**Prefer names from the business domain in which the software operates, not from the technical names that come from the programming language: use CompanyNames rather than StringArray.**

It would be useful if Boolean identifiers start with **is**, **has** or **can** (with an uppercase letter for properties), but only if this adds for clarity.

Constants follow next. They should be ordered according to their access modifier – **public** constants are first, then **protected** and then **private!**

After non-static class fields, **constructor declarations** follow. After the **constructors**, **properties** are declared.

**Abstraction**

**Def:**

A few basic rules:

- Public properties of a class should have the same level of abstraction.

- The interface of a class should be simple and clear.

- A class should describe only one thing.

- A class should hide its internal implementation.

Code is developed and changes and evolves over time. In spite of the evolution of classes, their interfaces should remain in-tact. A **bad practice** of a class having **inconsistent interface** is shown below:

class Employee

{

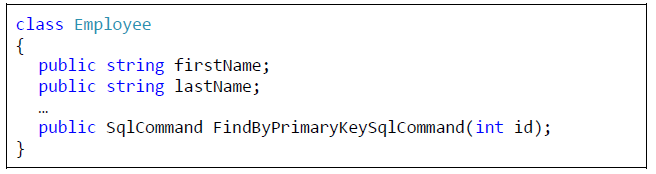
public string firstName;

public string lastName;

…

public SqlCommand FindByPrimaryKeySqlCommand(int id);

}

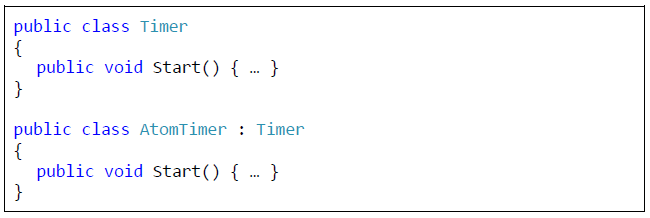


The latter method is incompatible with the level of abstraction at which **Employee** works. The user of this class should not be aware at all that a database is used internally.

**Inheritance**

**Info:**

**Do not hide methods** in derived classes:



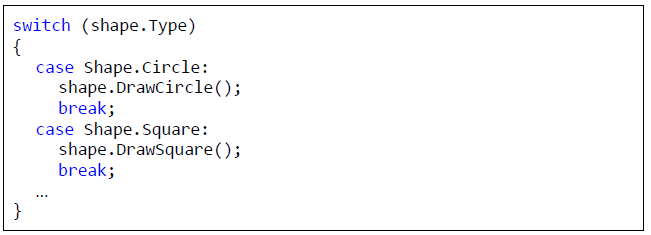
The method in the derived class **hides** the base (original) implementation. This is **not recommended**. If, in a rare case, this is desired and necessary, the keyword **new** should be used.

Move common methods, data and behavior as high as possible in the inheritance tree. This way, functionality is less likely to be duplicated and will be accessible to a wider audience.

If you have a class with a **single successor only**, consider this suspicious. That level of abstraction is probably unnecessary. A suspicious method would be one that re-implements a base method, but does nothing more than the corresponding base method.

Deep inheritance with **more than 6 levels** is hard for tracing, debugging and maintaining, and is **not recommended**. In a derived class, use membervariables through properties, rather than directly.

The example below demonstrates **wrongly written code** when inheritance should be preferred over type checking:



It would make more sense if **Shape** was inherited by **Circle** and **Square**, which implement the virtual method **Shape.Draw()**.

**Encapsulation**

**Info:**

A good approach is to make all members **private**. Only those of them that should be visible from outside could be marked **protected**, or eventually **public**.

**Implementation details should be hidden**. The user of a high-quality class should not be aware of its inner-workings; he should only know what it does and how it is used.

Member-variables (fields) **should be hidden** behind properties. Public member-variables are a manifestation of low-quality code. Constants are an exception in this regard.

The public members of a class should be **consistent** with the abstraction represented by this class. Do not make assumptions about the usage scenario of a class.

**Do not rely on undocumented, internal implementation logic.**

**Constructors**

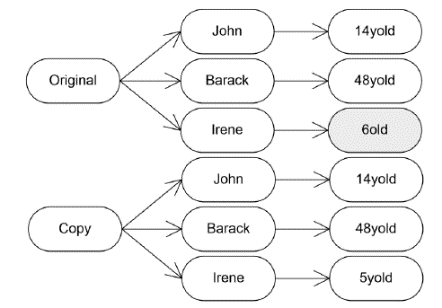
**Info:**

It is preferred that all class members are **initialized in the constructor**. Usage of an uninitialized class is dangerous. A half-initialized class is maybe even more dangerous. Initialize member-variables in the same order as they are declared.

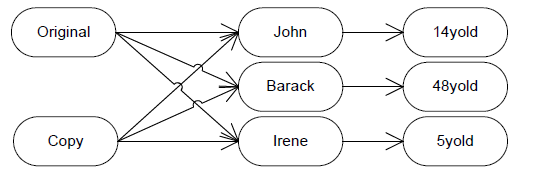
**Deep and Shallow Copy**

**Info:**

When we assign values sometime we need to copy an object (make a duplicate). This can be done in two ways: **deep copy** or **shallow copy**.

**Deep copies** of an object are copies in which all member-variables are copied, and their member-variables also, and so on, until no other member-variables refer to objects. In a shallow copy, only the members at the first level are copied. **Example of deep copied object** and its members:

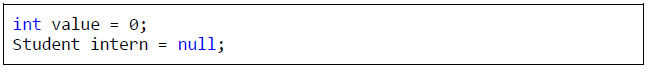
**Shallow copies** work differently. When a shallow copy is created, the original object and its copy **share some of their members**:



**Shallow copies are dangerous** because a change in one object leads to indirect changes in others. Notice how the change of Iren’s age in the original does not affect the age of Iren in the copy when we use deep copies. With shallow copies, the change will be reflected in both places.

**A public method should either correctly accomplish exactly what its name suggests, or should inform the caller for an error by throwing an exception.**  **Any other behavior is incorrect!**

A good practice, however, is to initialize all variables explicitly at the time of their declaration:



**Scope, Lifetime and Span of Variables**

**Info:**

The term **variable scope** actually denotes **how “famous” a variable is**. In .NET, three layers of variable scope exist: **static** variables, member-variables of a class (**fields**), and **local variables** inside a method.

If you don’t know what scope to use, start with **private** and if needed, switch to **protected** or **public**.

It is important that the programmer tracks the usage of a particular variable, along with its scope, span and lifetime. The main objective is to **reduce the scope, the lifetime and the span** as much as possible.

**Declare local variables as late as possible, immediately before using them for the first time. Initialize them at the time of declaration!**

**Use of Constants**

**Info:**

Well written code should not contain “**magic numbers**” and “**magic strings**”. Such constants are all the literals in a program having a value other than **0**, **1**, **-1**, **""** and **null** (with little exceptions).

It comes to mind that it is better to define the repeating values only once on the code. In .NET such values are declared as **named constants**.

Constants should generally be defined for every number or string that is used more than once in a program (with some exceptions).

When a constant does not contribute to the readability of the code, you should avoid it.

**Use named constants to avoid the usage and duplication of magic numbers and strings, and mostly to improve code readability.**

**If the introduction of a named constant hinders the readability, better leave the hardcoded value in the code!**

**Proper Use of Control Flow Statements**

**Info:**

An **if**-statement without curly brackets only takes the first statement as its body, regardless of the indentation, which makes matters confusing.

**Always enclose the body of loops and conditional statements in curly brackets – { and }.**

Extracting parts of the code into **separate methods** is the easiest and most efficient way to reduce the level of nesting of a group of conditional statements, while preserving their logic.

**Proper Use of Loops**

**Info:**

If we need a loop that will execute a **fixed number of times**, a **for**-loop is a good fit. This kind of loop is used in the most basic situations where interrupting the control is not necessary.

The initialization, the check of the condition and the incrementing are all in the **for**-construct and the loop body does not care about that. The value of the counter should not be altered within the body.

If it is necessary to check **some conditions in order to stop** the execution of the loop, then it is probably better to pick a **while** loop. A **while** loop is suitable in cases where the exact number of iterations is not known.

The execution there continues until the exit condition has been encountered. If the prerequisites for using a **while** loop are in place, but the loop body must unconditionally **execute at least once**, a **do-while** loop should be used instead.

**Assertions vs. Exceptions**

**Info:**

**Exceptions are announcements for an error** or for an unexpected event. They inform the programmer using the code for an error. Exceptions can be caught and program execution can still continue.

**Assertions produce fatal errors**. They cannot be caught or handled, because they are meant to indicate a bug in the code. A failed assertion causes the program to terminate.

**Assertions can be turned off**. The concept is to have them turned on only at the time of developing, in order to find as many bugs as possible. When turned off, the conditions are no longer checked.

Turning off the assertions is plausible when the software goes to production, since these checks are affecting the performance and the messages are not always meaningful to the end user.

**If a particular check should continue to exist when the software goes to production (for example, checking the input that comes from the user), it should not be implemented as an assertion in the first place. Exceptions should be used in such cases instead.**

**Assertions should only be used for conditions that, if not met, it is due to a bug in the program.**

**Lambda Expressions and LINQ**

**Extension Methods**

**Info:**

In practice, programmers often have to **add new functionality** to already existing code. If the code is available, we can simply add the required functionality and recompile. When a given assembly (**.exe** or **.dll** file) has already been compiled, and the source code is not available, a common way to extend the functionality of the types is trough inheritance.

This approach can be quite difficult to apply, due to the fact that we will have to change the instances of the base class with the instances of the derived one to be able to use our new functionality. Unfortunately, that is the least of our problems.

If the type we want to inherit is marked with the keyword **sealed**, inheritance is not possible.

**Extension methods** solve that very same problem – they present to us the opportunity to **add new functionality to already existing type** (class or interface), without having to change its original code or use inheritance, i.e. also works fine with types that cannot be inherited. Notice that trough extension methods we can add “implemented methods” even to interfaces.

**The extension methods** are defined as **static** in ordinary static classes. The type of their first argument is the class (or the interface) they extend. In front of it, we should place **the keyword this**.

That is what makes them different from other static methods, and indicates the compiler that this is an extension method.

The parameter with the keyword **this** in front of it can be used in the method body to create its functionality. Practically, it is the object that is used by the extension method.

Extension methods can be applied directly to objects of the class/interface they extend. They can also be invoked statically through the static class they are defined in, but it is not a good practice.

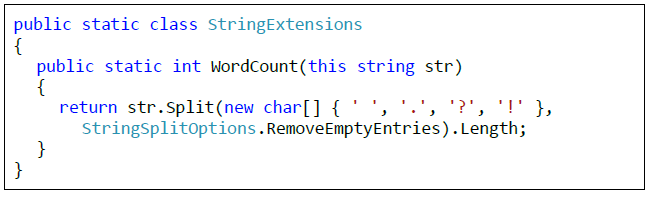
**To refer to a specific extension method, we should add “using” and the corresponding namespace, where the static class, describing this method, is defined. Otherwise the compiler has no way of knowing about their existence!**

Have in mind, that the type **string** is **sealed**, so it cannot be inherited.

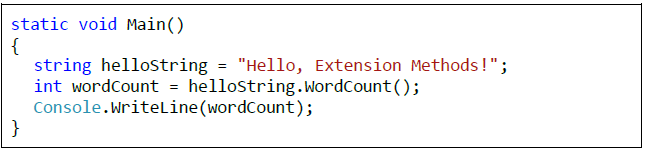
**Extension Methods – Examples**

**Info:**

Let’s take for example the **definition of an extension method** that counts the number of words in a given string. Have in mind, that the type **string** is **sealed**, so it cannot be inherited.



The method **WordCount(…)** extends the class **String**. This is indicated by the keyword **this** before the type and the name of the first argument of the method (in our case **str**).

The method itself is **static** and it is defined in the static class **StringExtensions**. The usage of the extension method is done the same way as all the other methods of the class **String**. Do not forget to add the corresponding namespace, where the static class, describing the extension methods, is defined. Example of **using an extension method**:

The method is invoked on the

object **helloString**, which is of

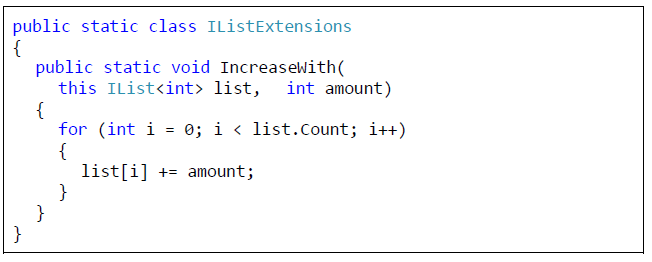
type **string**. It also takes the

object as an argument and works with it (in our case refers to its **Split(…)** method and returns the number of elements of the array, produced by the **Split(…)** method).

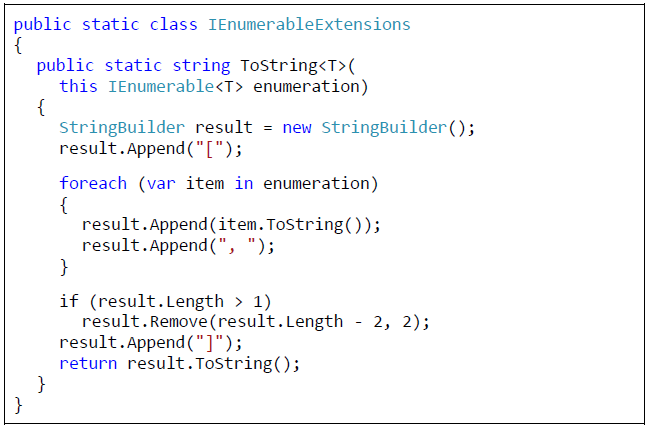
**Extension Methods for Interfaces**

**Info:**

Extension methods can not only be used on classes, but on interfaces as well. Our next example takes an instance of a class, that implements the interface list of integers (**IList<int>**), and increases their value by a certain number.

The method **IncreaseWith(…)** can access only those elements that are included in the interface **IList** (e.g. the property **Count**).

The extension methods also give us the opportunity to work on generic types. Let’s take for example a method that loops trough a collection, using **foreach**, implementing **IEnumerable** from generic type **T**. Its purpose is to convert to a meaningful string a sequence of elements (e.g. a list of integers):



**Anonymous Types**

**Info:**

In object-oriented languages (such as C#), it is common to define small classes that will be used only once. Typical example is the class **Point** that has only two fields – the coordinates of a point.

Creating a simple class with the idea of using it just once is inconvenient and time consuming for the programmer, especially when the standard operations for each class: **ToString()**, **Equals()** and **GetHashCode()** have to be predefined.

In C# there is a built-in way to create **single-use types**, called **anonymous types**. Objects of such type are created almost the same way as other objects in C#. The thing with them is that we don’t need to define data type for the variable in advance.

The **keyword var** indicates to the compiler that the type of the variable will be automatically detected by the expression, after the equals sign.

We actually don’t have a choice here, since we can’t tell the specific type of the variable, because it is defined as one of an **anonymous** **type**. After that, we specify name for the object, followed by the "**=**" operator and the keyword **new**. In curly braces we enumerate the names and the values of the properties of the anonymous type.

**Anonymous Types – Example**

**Info:**

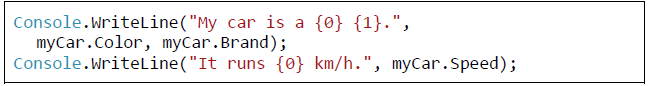
Here is an **example of creating an anonymous type** that describes a car:



During compilation, the compiler will create a class with a **unique name** (something like **<>f\_\_AnonymousType0**) and will generate properties for it(with getter and setter).

In the example above, the compiler will guess by itsown, that the properties **Color** and **Brand** are of type **string** and **Speed** willbe set as **int**.

Right after the initialization, the object of the anonymous typecan be used as one of an ordinary type with its three properties:



As any other type in .NET, the anonymous ones inherit the class **System.Object**. During compilation, the compiler will automatically redefine themethods **ToString()**, **Equals()** and **GetHashCode()** for us.

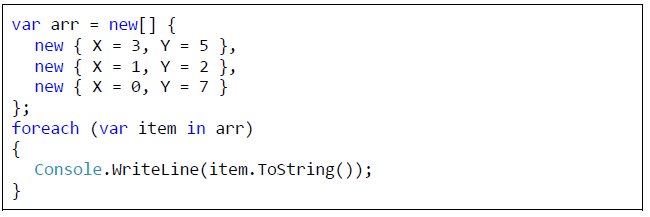
**Arrays of Anonymous Types**

**Info:**

The anonymous types, like ordinary ones, can be used as **elements of arrays**. We can initialize them with the keyword **new**, followed by squarebrackets. The values of the elements of the array are listed the same way, as the values assigned to the anonymous types.

The values in the array should be homogeneous, i.e. it is not possible to have different anonymous types in the same array.

An example of defining an array of anonymous types with two properties (**X** and **Y**):



**Lambda Expressions**

**Info:**

**Lambda expressions are anonymous functions** that contain expressions or sequence of operators. All lambda expressions use the lambda operator **=>**, which can be read as “**goes to**”.

The idea of the lambda expressions in C# is borrowed from the functional programming languages (e.g. **Haskell**, **Lisp**, **Scheme**, **F#** and others). The left side of the lambda operator specifies the **input parameters** and the right side holds an **expression** or a code block that works with the entry parameters and conceivably returns some result.

Usually lambda expressions are used as **predicates** or instead of **delegates** (a type that references a method instance), which can be applied oncollections, processing their elements and/or returning a certain result.

**Lambda Expressions – Examples**

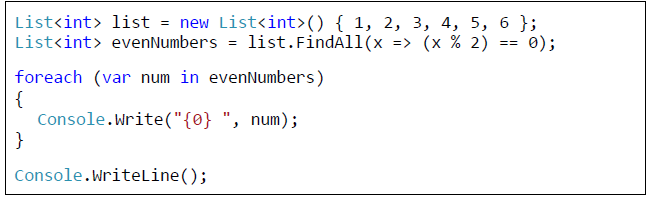
**Info:**

As an example, let’s take the **extension method FindAll(…)**, which can be used to filter the necessary elements. It works on a certain collection by applying a given **predicate** on it that checks if an element matches a certain requirement.

In order to use it we have to add a reference to the assembly

**System.Core.dll** (if it is not already added) and include the namespace **System.Linq**, because the extension methods for the collections are there.

For example, if we want to take only the even numbers from a collection of integers, we can use the method **FindAll(…)** on that collection, passing a lambda method to it that checks if a certain number is even:



**Using Lambda Expressions with Anonymous Types**

**Info:**

We can create **collections of anonymous types** from a collection with some elements by **using lambda expressions**. Let’s take the collection **dogs**, containing elements of type **Dog**, and create new collection consisting of elements of an anonymous type, having two properties – age and the initial letter of the dog’s name:

**LINQ Queries**

**Info:**

**LINQ (Language-Integrated Query)** is a set of extensions of the .NET Framework, that includes language integrated queries and operations on the elements of a certain **data source** (most often arrays or collections).

LINQ is a **very powerful tool**, similar to most SQL languages by logic and syntax. It actually works with collections in the same way as SQL languages work with table rows in databases. It is part of the syntax of C# and Visual Basic .NET and consists of few special keywords like **from**, **in** and **select**.

In order to use LINQ queries in C#, we have to include a **reference to System.Core.dll** and to **include the namespace System.Linq** in the beginning of the C# program.

**Data Sources with LINQ**

**Info:**

To define the data source (collection, array and so on), we have to use the keywords **from** and **in** and a variable for the iteration of the collection (the iteration is similar to the one with the **foreach** operator). For example, a query that starts like this:



can be read as follows: "for each element of the collection **CultureInfo.GetCultures(CultureTypes.AllCultures)** assign the variable **culture** anduse it to refer to these items further in the query".

**Data Filtering with LINQ**

**Info:**

The keyword **where** can be used to set conditions, that should be kept by each item of the collection, in order to continue with the execution of the query.

The expression after **where** is always of a Boolean type. We can say that **where works as a filter for the elements**. For example, if we want to see only those cultures, whose name begins with the lowercase Latin letter **b**, we can continue the query from our last example like this:

As we can notice, after **where … in**, we use only the name we gave for the iteration of the variables in the collection. The keyword **where** is compiled up to the invoking of the extension method **Where()**.



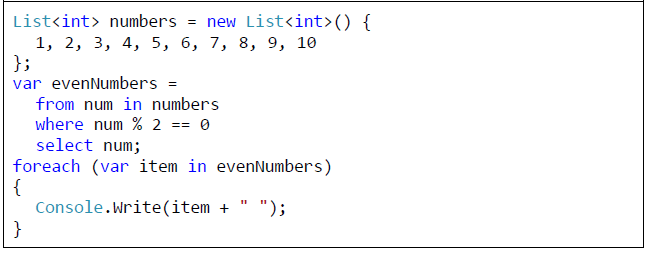
**Results of LINQ Queries**

**Info:**

To **choose the output data** for the query, we can **use the keyword select**. The result is an object of an existing class or an anonymous type.

The result can also be a property of the objects, the query runs through or the objects themselves. The **select** statement and everything following it is placed **always at the end of the query**.

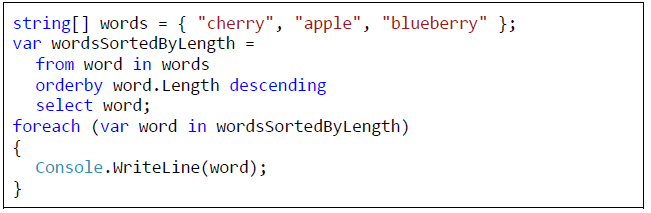
The four keywords: **from**, **in**, **where** and **select**, are completely enough to create a simple LINQ query. Here is an example:



**Sorting Data with LINQ**

**Info:**

**Sorting with LINQ queries** is done through the **keyword orderby**. The conditions, used for sorting the elements, are placed after it. For each condition the order of arrangement can be indicated: ascending (using the keyword **ascending**) and descending (with the keyword **descending**), as by default the elements are ordered in ascending order. If we want to sort an array of strings by their length in descending order, for example, we can write the following query:



**Grouping Results with LINQ**

**Info:**

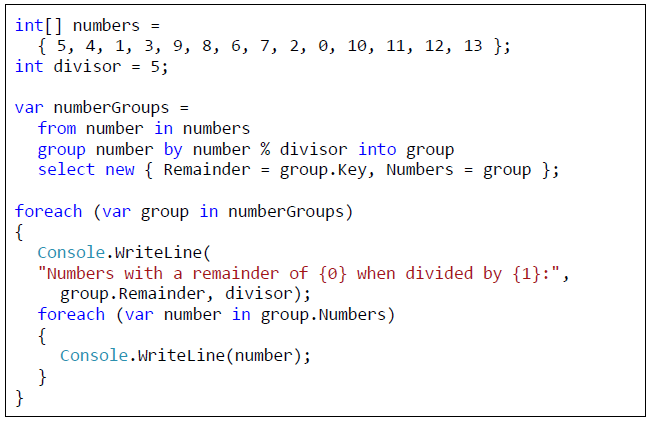
To group the results by some criteria the keyword **group** should be used. The pattern is as follows:



The **result of grouping is a new collection of a special type** that can be used further in the query.

After the grouping, however, the query stops working with its initial variable.

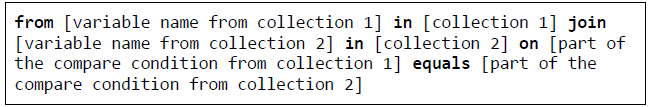
This means that in the **select** statement, we can use only the group. An example of grouping:



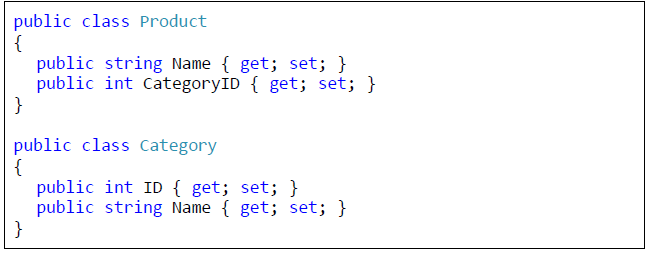
**Joining Data with LINQ**

**Info:**

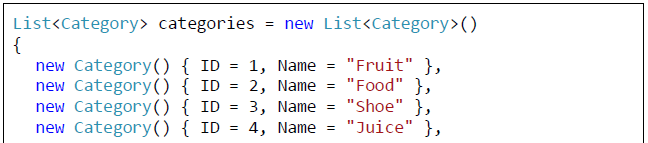
The **join** statement is a bit more complicated than the other LINQ statements. It joins collections by certain matching criteria and extracts the needed data. Its syntax is as follows:



Further in the query (e.g. in the **select** part), both, the name of the variable from collection 1, and the name of the variable from collection 2, can be used. Example:



The code that illustrates how to use LINQ joins:

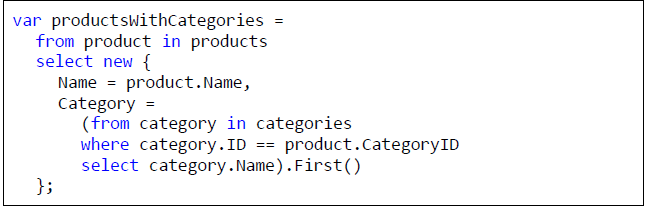




**Nested LINQ Queries**

**Info:**

LINQ also supports **nested queries**. For example our last query can be written by nesting two queries in the following way (the result is exactly the same as the one with **join**):



Since each query in LINQ returns a collection of items (irrespective of whether the result from it is of 0, 1 or more elements), we need to use the extension method **First()** over the result of the nested query.

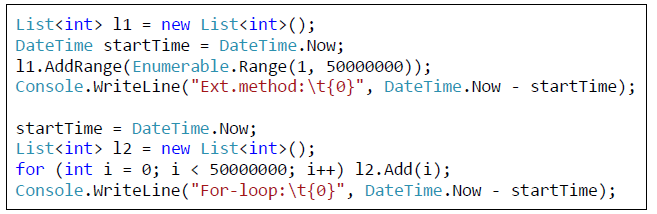
The method **First()** returns the first element (in our case the only one) of the collection it is applied on. In this way we get the name of the category only by its ID number.

**LINQ Performance**

**Info:**

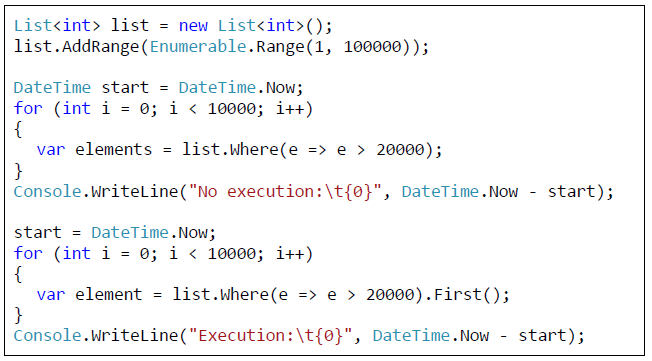
As a rule using **LINQ and extension methods is slower than using direct operations** over a collection of elements, so beware of using LINQ whenprocessing large collections or the performance is critical.

Let’s compare the speed of adding 50,000,000 elements to a list through extension methods and directly with a **for**-loop:



LINQ technology and extension methods work through the concept of **expression trees**. Each LINQ query is translated by the compiler to an expression tree and is executed when its results are actually accessed (not earlier).

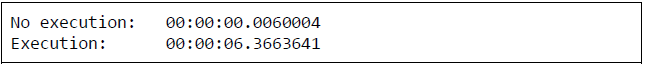
For example let’s consider the following code:



This shows that if we call a **.Where(…)** filter (or **where** clause in LINQ) it is not actually executed until its result is actually needed. The elements **get filtered** **on demand**, at the time they are really required. In our case this is when we invoke **First()** method.

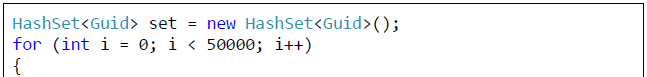
Moreover, if we get the first element of a sequence, the rest elements are not processes until needed. Thus if we use change the

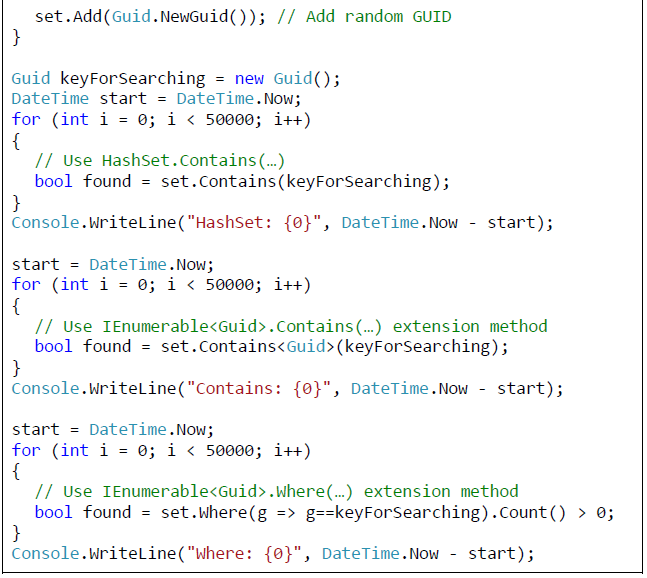
filtering lambda function from “**e => e > 20000**” to “**e => e > 500000**”, the filtering becomes times slower because more elements are processed until the first matching the filtering condition is found:



Standard .NET Framework collection classes like **List<T>**, **HashSet<T>** and **Dictionary<K,V>** are optimized to work fast with LINQ. Most operations with LINQ work almost as fast as if we run them directly.

Let’s check this example:





Seems like .NET Framework takes into account the capability to search in constant time O(1) in a **HashSet<T>**, so searching though the native metho **Contains(…)** and though the extension methods **IEnumerable.Contains(…)** both **run in time O(1)**.

By contrast, the **IEnumerable.Where(…)** method is **dramatically slower** and runs in linear time O(n). This is expected, because the **Where(…)** method checks certain condition for each element in a collection and it is expected to process all elements one by one. By contrast the **Contains(…)** method just searches for single element which is fast operation.

In the above example we use the system structure **Guid**. This is a global unique identifier often used in computer technologies to identify an object. It may look like the following: **8668f585-faf8-4685-8025-6a8d1d2aba0a**. If you want to generate a global unique (world-wide) identifier, you might benefit from the method **Guid.NewGuid()**, like we do in the code above.